An Intelligent Component Database For Behavioral Synthesis

BY

Gwo-Dong Chen
Daniel D. Gajski

Technical Report 89-39

Information and Computer Science
University of California at Irvine
Irvine, CA 92717

Abstract

This paper describes an intelligent component database system that delivers components to synthesis tools when given a set of attributes and constraints. Requirements of a component server are defined and an implementation is described. Our experiments demonstrate that such a component sever can replace component libraries and component catalogs with hundreds of pages.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

A high level synthesis system requires multiple tools to convert a behavioral description into layout. For instance, one tool is needed to take the behavioral description and generate a microarchitecture structure. Other tools are required to convert this initial structure into layout. All of these tools require a component database. The component database generates components for synthesis tools that fit specific design requirements, and provides information about a component's area, delay, layout characteristics (shape function, footprint, etc.), and possible tradeoffs. This information is essential for synthesis tools to perform technology mapping and examine various tradeoffs. Thus, in addition to generating components, the component database provides all necessary information for design exploration.

Existing behavioral synthesis systems have focused on two approaches for component databases: use of a fixed component library [JKMP89] or use of a generic component library [ThDW87][TrDi89]. When using a generic library, a synthesis tool does not have information on the component's delay or area. Thus the synthesis tool can not make proper tradeoffs. A similar problem exists at the layout level. To generate the layout, the floorplanner makes use of the component's shape function. A shape function produces a range of available aspect ratios (ratio of width to height) for a component. By having more aspect

ratios available, a layout tool can produce a better floorplan. Generic libraries, however, can not provide a shape function. Thus, generic libraries fail to provide synthesis tools with enough information to make good design tradeoffs.

When using a fixed component library, it is also difficult to generate optimal designs. The synthesis algorithm may be forced to use a component that does not fit the design constraints. For example, if a component has an active high input, the synthesis tool must either (1) add an inverter or (2) design and generate a new component instead of using the component database. Sometimes, the speed of a component is slower than required. If the synthesis system can not generate its own faster component, it must increase the delay constraint, sacrificing performance. Further, the library must be rebuilt in the case of a technology change. An additional limitation of the fixed component library occurs in the floorplanning stage. The floorplanner may not have enough component shape alternatives to obtain minimal area. For example, in some cases, there may be enough space to place a component but the component shape may not fit into the available space.

These problems with traditional approaches indicate that design synthesis tools need an intelligent component database that can dynamically generate components for different constraints when given a set of component attributes. In addition, synthesis tools need accurate delay and area estimates for design

exploration. Previous work [Mcfa86] [Mcfa87] has shown that good estimates are crucial for evaluating a design. However, layout tools can take hours to generate a component layout and require a large amount of disk space to store the layouts. To avoid these problems during design exploration, the database must have tools that can quickly estimate a component's delay, area, shape, and power consumption.

There are three basic requirements for an effective component server: (1) an internal component representation and description, (2) a user interface to the component server, and (3) tools for component generation and parameter estimation. A **language** that describes component implementations should be able to (1) describe the behavior of all microarchitecture components and (2) describe a component with different attributes (such as active low/high input, tri-state output, etc.). The **user interface** is required to request component generation and to query about available components and their attributes. **Tools** should be provided to generate components, to estimate the delay and shape function, and to verify that generated components are correct and meet the given constraints.

Previous systems have been reported that provide a set of fixed module generators with limited parameters [ChMa88] [CoSn88] [RDVG88]. These systems provide an interactive interface to retrieve information from these

generators. In these systems, a list of basic cells were first laid out by hand. Then, a language was provided to assemble these basic cells and to create parameterized module generators. The port positions, delays, and shapes of each component were fixed. With such an approach designing a new module generator is a time consuming task. Information about these modules is embedded in the synthesis system, making it difficult to insert new components. Basically, these systems retain the disadvantages of a fixed component library. Fred [Wayn86] is a system that answers queries about abstract components described in the Ethel language. However, Fred does not generate components from an Ethel description.

In this paper, a component server, ICDB (Intelligent Component DataBase), is described. ICDB can dynamically generate components for given set of constraints and attributes. Further, it provides delay, area and shape estimates that support design tradeoffs on the microarchitecture level. By providing more flexible components than previous component databases and by providing more synthesis parameters besides just delay and area, ICDB allows better exploration of design alternatives and hence generation of higher quality designs.

The rest of this paper is organized as follows. Section 2 provides an overview of the database. The language used to describe a component

implementation and the user interface language are explained in Section 3. Section 4 describes the component management and component generation of ICDB. Finally several examples are given in Section 5 that demonstrate the capabilities of this component server.

2. System Overview

2.1. The role of ICDB in behavioral synthesis

The component database system ICDB is used as a component server for different behavioral synthesis tools. Figure 1 depicts the role of ICDB in the behavioral synthesis system. It shows that ICDB is used in three phases of synthesis: behavioral synthesis, partitioning and optimization, and floor planning and layout generation.

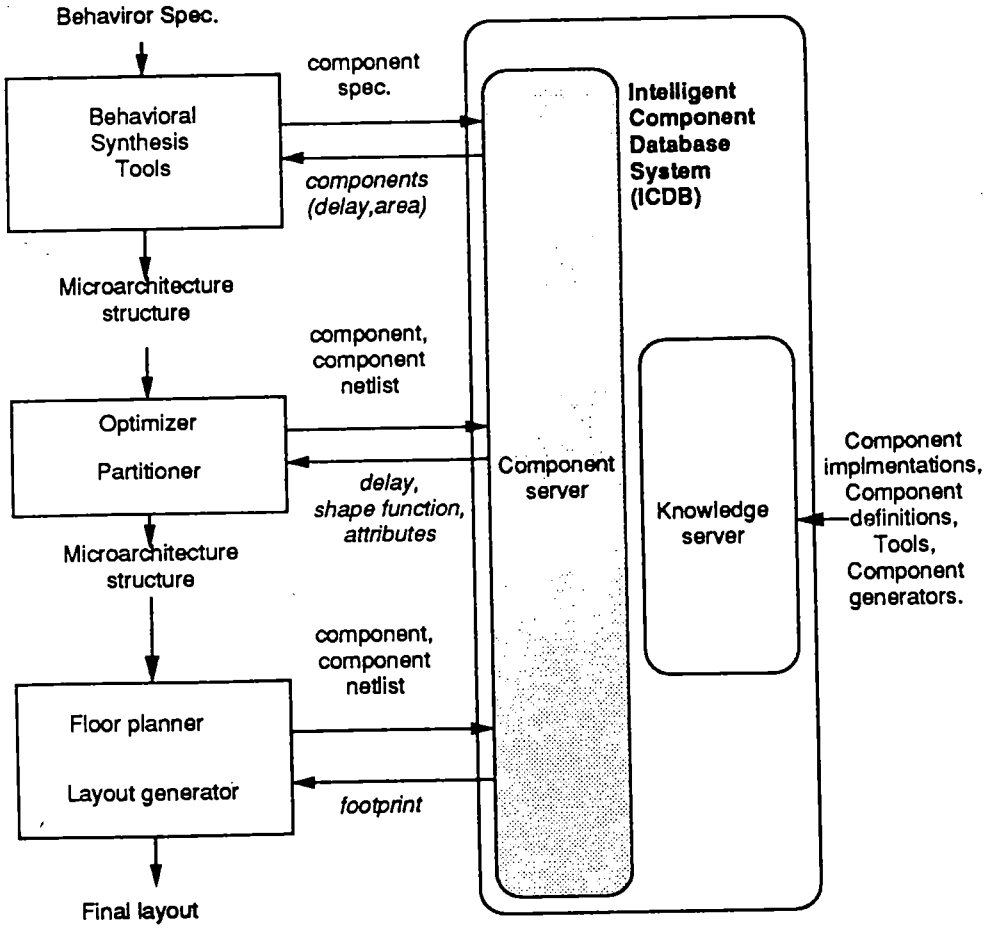


Figure 1 Functions of ICDB

During the process of transforming a behavioral description into a microarchitecture level structure, ICDB provides the delay, minimum clock width, area, and minimum setup and hold time for each component. During operator scheduling, a synthesis tool can use the component delay time to

determine the proper clock width. A behavioral synthesis tool can also use the information to decide whether to chain two operations together in a single clock, or whether to place an operation in a multiple clock step. When doing resource allocation, ICDB informs the synthesis tool which components perform the requested functions. Thus, the tools can select appropriate components according to the delay requirements. In later design stages, tools can replace these selected components with other components that better meet additional considerations, such as area shape for floorplanning. In the microarchitecture optimization phase, ICDB is queried to determine if components can be merged and whether merging can produce a better design. For example, a register and a incrementer can be merged into a counter.

To achieve a good floor plan, the partitioner can try different ways of clustering components and retrieve their shape function from ICDB. It can also assign the pin positions of the combined object and ask ICDB to generate the layout accordingly.

2.2. User's view of ICDB

ICDB is composed of two subsystems: (1) a knowledge acquisition support system and (2) a component server. ICDB provides a Component Query Language (CQL) as a user interface for both subsystems. The user's view of ICDB is shown in Figure 2.

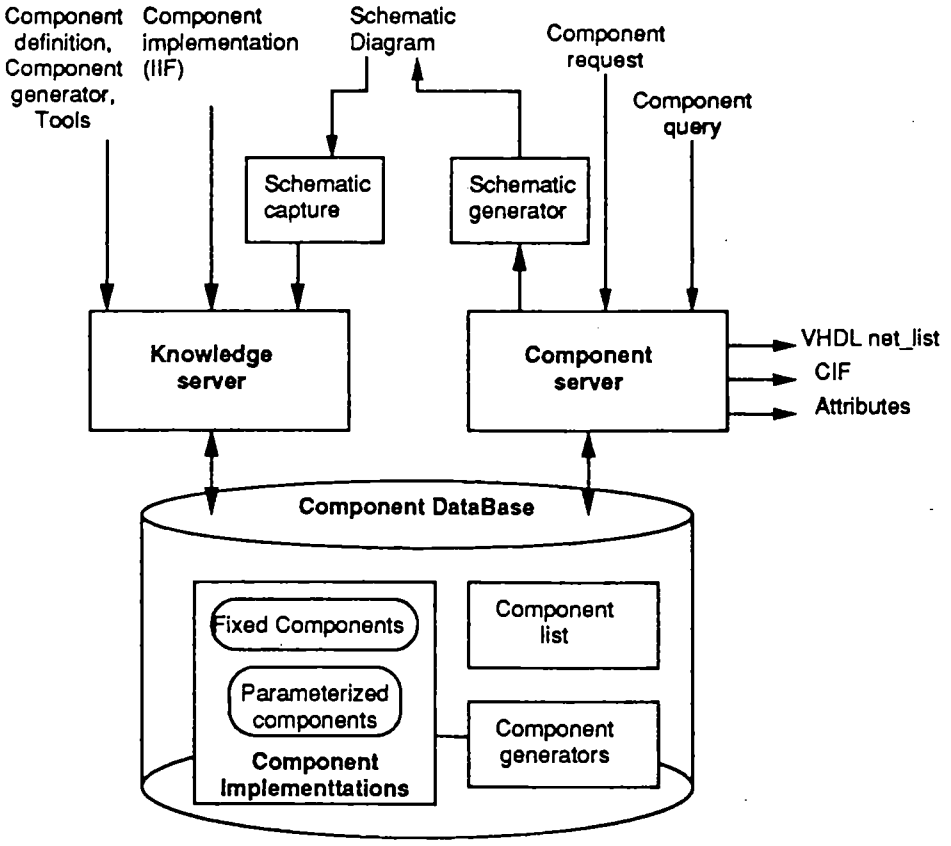


Figure 2 User's view of ICDB

Users can insert component definitions, component generators, tools, and component implementations to ICDB through the knowledge acquisition support mechanism. An intermediate format called IIF (described in section 4) is used to

describe a component's implementation.

There are two types of component implementations: (1) fixed components and (2) parameterized components. The component generators can generate components from parameterizable component descriptions. For example, there may be an n-bit adder described in IIF, a 4 bit adder stored in CIF format, and an adder module generator which can generate adders from the IIF description of an n-bit adder.

During the synthesis process, many components will be generated. The required components are stored in a component list. Both the component specifications and the components generated are stored. Thus, these components do not need to be regenerated. Information about each component can be communicated between tools through ICDB.

The component server provides two types of facilities. It (1) generates components from a given component specification and (2) answers queries about component implementations and generated components. A generated component is represented in a VHDL netlist for logic level structure or CIF for layout. ICDB provides the area, delay, and attributes of all generated components.

2.3. System Architecture of ICDB

The system architecture of ICDB is shown in Figure 3.

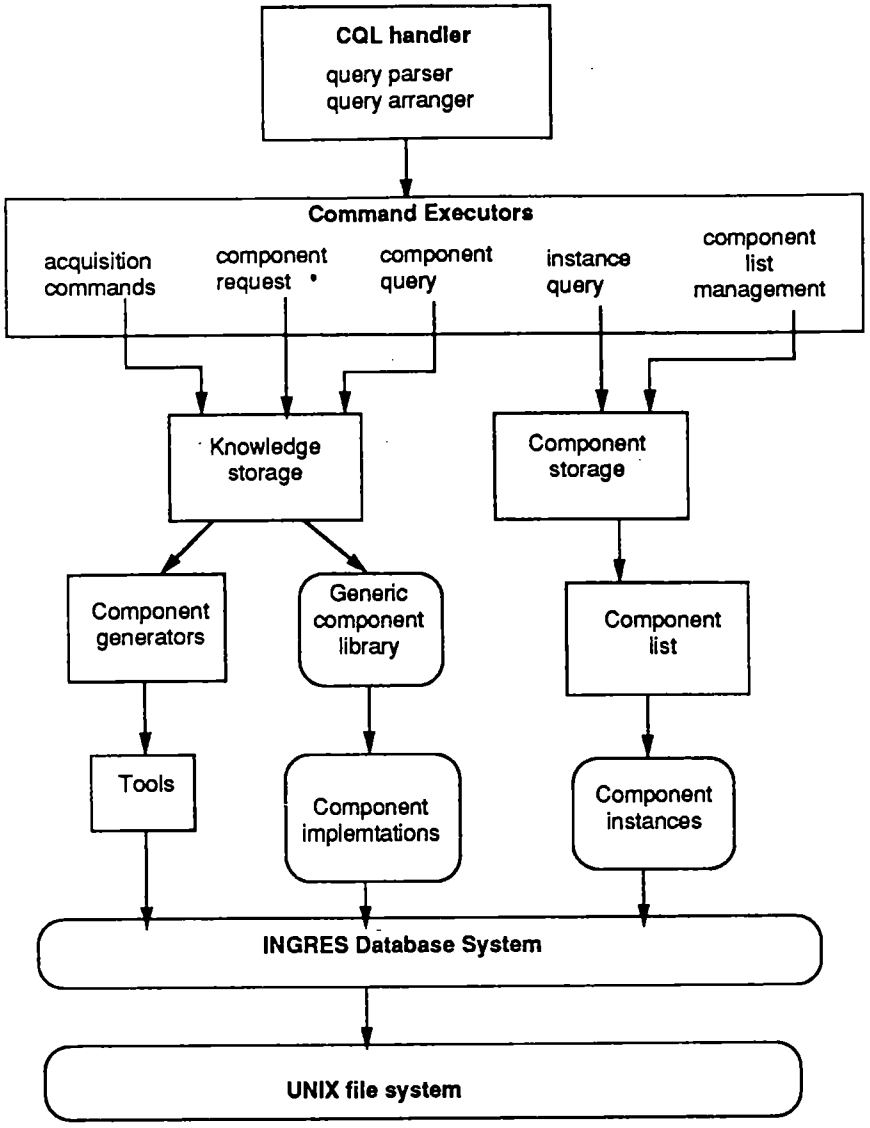


Figure 3 System architecture of ICDB

ICDB has a CQL query parser and a query handler. Each CQL command has a corresponding program to execute it. The component generation manager consists of two parts: (1) a component generator manager and (2) a generic component library. The generic component library stores ICDB component implementations and answers queries about them. Components described in any description format can be stored in the generic component library. When a user asks ICDB to generate components, the generic component library will be queried to answer what components are available. In addition, the library will provide the information used to connect the component with other components, such as port type, equivalent ports, and inverted ports.

The data used to describe component generators, component definitions, component implementations, and generated components is stored in the INGRES database system. ICDB uses SQL to query this data from INGRES. The component design data is stored in the UNIX file system. Tools communicate directly with the UNIX file system. They retrieve the UNIX file name from ICDB, then perform their own I/O. Thus, ICDB does not degrade the performance of tools.

Component database tools such as logic synthesis and optimization tools [BrRu87] [Ke87] [BeOw87] [Vaga88], transistor sizing tools [FiDu85] [Hedl87], layout tools [LiGa88] [ChCh89] [OnLL89], and estimation tools are stored in the

tool library. Tools are organized as a list of component generators. A component generator is composed of (1) tools to generate components from given attributes and constraints, and (2) tools to produce information about generated components such as area, delay, and shape of the components they generate. A component can be generated by different component generators, providing a wider variety of components. The component generation manager will arrange the generation sequence and answer component queries.

Many components will be generated from the given specifications. These components, their specification, and their description are stored and managed by the component manager. Thus, these components can later be refined and modified during the synthesis process.

3. ICDB user interface

ICDB provides two user interface languages: (1) an intermediate format (IIF) to describe the low-level behavior of components and (2) a Component Query Language (CQL) to query what components are available and to query component characteristics such as area, delay, shape function, and footprint.

3.1. Component implementation description

In order to describe microarchitecture level components (such as counters, shift registers, etc.), we need a format capable of describing sequential,

asynchronous behavior, and I/O conversion. We define an Intermediate Format (IIF) which extends a boolean expression language with clocking and asynchronous behavior in order to describe generic components composed of logic gates, flip-flops with asynchronous set and reset, and interface components. In addition, IIF provides programming structures for describing parameterized objects. The programming structures include IF, FOR loop, and function call statements that enable designers to specify a component with replicatable structure.

IIF is an extension of the Berkeley EQN (equation) format for describing boolean expressions. Besides providing the basic boolean operations of AND, OR, NOT, XOR, and XNOR, IIF contains operators for specifying a D flip-flop with asynchronous set and reset, and operators for tri-state, delay, schmitt trigger, and wire-or.

In order to describe components with different bit widths, IIF must be extended with additional language constructs. For example, in designing a 16-bit adder, we may start with a one bit adder, then replicate it sixteen times to build the 16-bit adder. Sometimes, we use existing components to build other components. For instance, we can use an adder and exclusive OR gates to build an adder/subtractor. Therefore, IIF provides facilities for using a replicatable structure to describe parameterized components.

IIF has the same block structure and the similar syntax as the C programming language. There are three programming constructs in IIF: sequence, decision, and loop. The sequence structure is a list of statements enclosed by { and }. The **IF** statement is used for decision and the **FOR** statement for looping. Programming language theory proves that any structure can be described with three basic structures: sequence, decision, and loop. So IIF is complete from the structural point of view. A detail description of IIF is described in Appendix A.

An n-bit counter with parameters is described in the following example. IIF programming structures are used to describe different options for this counter. The **for** structure is used to construct an n-bit counter from a one-bit counter. The first **if** structure enables a user to select different architecture styles (ripple counter or synchronous counter). The ripple counter is called by an IIF function call. The **if** structure is also used to describe different options for this counter. It includes an optional **ENABLE** control, an optional asynchronous parallel load, and a choice of up count only, down count only, or both an up count and a down count. This example showns that IIF enables a user to easily specify a parameterized component.

```
NAME : COUNTER ;
PARAMETER : size, type, load, enable, up_or_down ;
INORDER : D[size], CLK, LOAD, ENA, DWUP ;
OUTORDER : Q[size], MINMAX, RCLK ;
PIIFVARIABLE : C[size+1], OVFUNF, CLK0 ;
SUBFUNCTION : RIPPLE_COUNTER;
VARIABLE : i, ripple_counter;
{
#cline ripple_counter = 1;
#if ( type == ripple_counter )   #RIPPLE_COUNTER(size);
#else   /* a synchronous counter */
 {
 C[0] = 1;

 #if( !enable) CLK0 = CLK;
 #else CLK0 = CLK@(~l ENA);


 #for(i=0;i<size;i++)
  {
   #if ( up_or_down == 1) C[i+1]= C[i] *  Q[i]; /* up counter only */
   #else #if ( up_or_down ==2) C[i+1] = C[i] * !Q[i]; /* down counter only */
   #else C[i+1] = C[i] * (Q[i] (+)DWUP);   /* updown counter  */

   #if ( load )              /* with asynchronous parallel load */
              Q[i] = (Q[i](+)C[i]) @(~rCLK0)
              ~a(0/(!LOAD*!D[i]),1/(!LOAD*D[i]));
   #else
       Q[i] = (Q[i](+)C[i]) @(~rCLK0);
  }

 #if(i=size) OVFUNF = C[size];
 MINMAX = CLK*OVFUNF ;
 RCLK = CLK*OVFUNF + !OVFUNF ;
 }
}
```

A schematic diagram of this counter with parameter values (size=4, type=2, load=1, enable=1, up_or_down=3) is depicted in Figure 4.

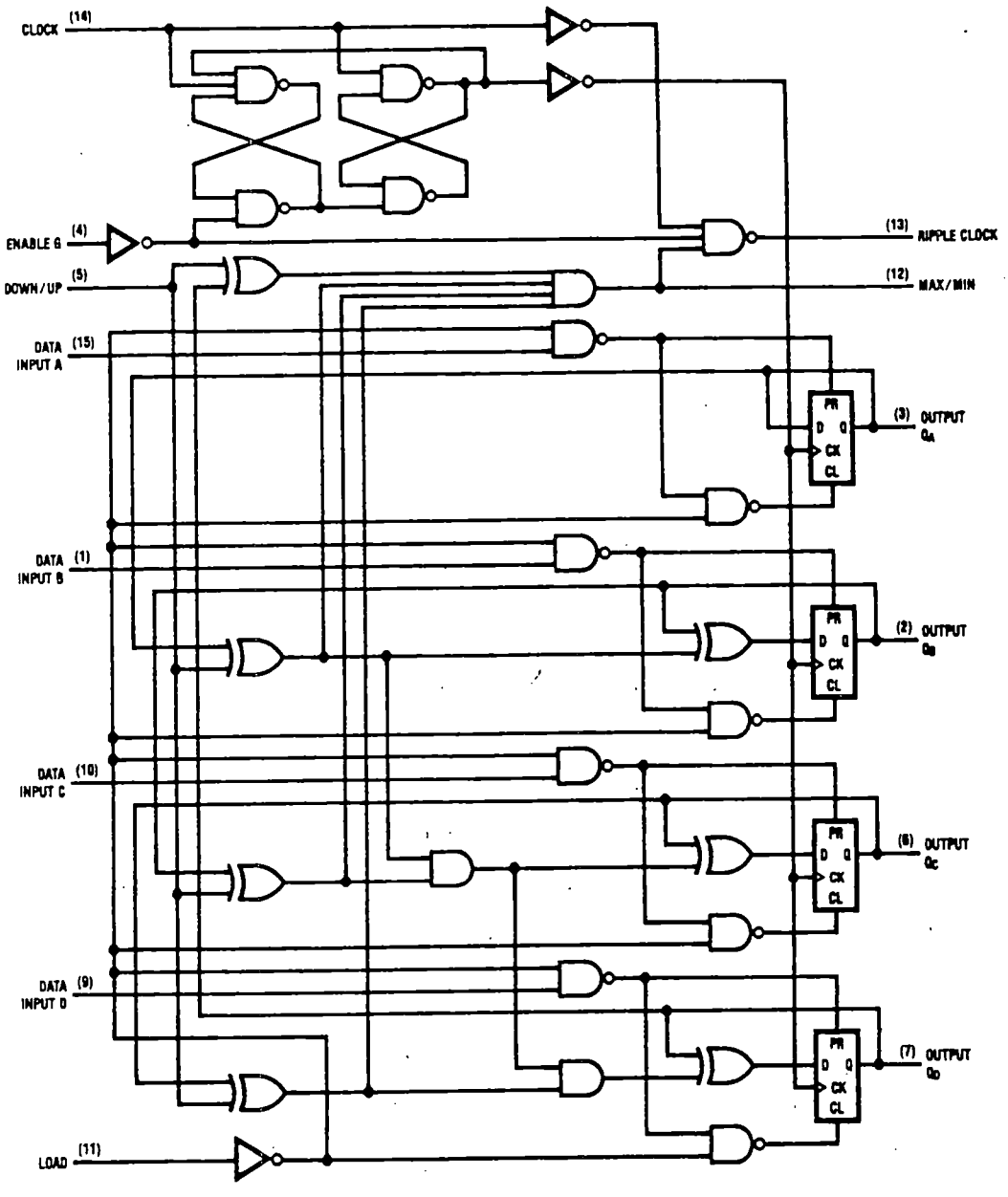


Figure 4 Logic schematic of a 4-bit updown counter

It is the standard TTL 74191 four-bit counter. In this situation, it is built from a one bit counter and expressed in IIF as:

Q=(Q(+)Cin) @(~rCLK0) ~a(0/(!LOAD*!Din),1/(!LOAD*Din))

Cout=Cin*(Q(+)DWUP);

The (+) is the exclusive OR operation. The first statement has two parts: a synchronous operation and an asynchronous operation. The expression **Q=(Q(+)Cin) @(~rCLK0)** means that output Q of the flip-flop will change when the Cin input is 1 (denoted by **Q=Q(+)Cin**) at the rising edge of CLK0 (denoted by **@(~rCLK0)**). The asynchronous expression **~a (0/ (!LOAD*!Din), 1/ (!LOAD*Din)**) means that Q is set to 0 when the expression **(!LOAD*!Din)** is 1 and set to 1 when the expression **(!LOAD*Din)** is 1. Thus, the **Din** data is loaded to flip-flop asynchronously when the signal **LOAD** is 0.

When it is to be used as an up-counter (DWUP is 0), Cin is 1 if previous bits are all 1. If current bit cell is 1 and input Cin is 1, then the next bit cell should be changed at the next clock; that is, Cout should be 1. The scheme is represented by **Cout=Cin*(Q(+)DWUP)** in the second statement of the one bit counter.

The expression **CLK0 = CLK@(~l(ENA))** indicates that the **CLK** is latched (**~l** means latch at low level) only when ENA is 0 (the expression **(ENA)** is 0). Therefore, this counter will count only when ENA (enable) is 1.

The **#if ... #else ...** structure indicates that the output Cout of a bit cell is passed to the next cell. If it is the last cell Cout is connected to port OVFUNF of this component.

The **#for(..;..;..)** structure indicates that this bit cell is to be replicated **size** times.

3.2. Component Query Language (CQL)

The Component Query Language (CQL) is the language user interface for ICDB's component server. CQL consists of four kinds of commands: (1) component queries, (2) component requests, (3) component instance queries, and (4) component list management. The detail description of CQL can be founded in Appendix B.

The component query is used to query about component implementations in ICDB. A user can request the generation of a component using the component request command. ICDB will generate new components for each component request command. The component instance query can be used to get information about them.

3.2.1. Component query

To build a function-to-component implementation mapping list, a synthesis tool can use a component query. For example, the following query can be used

to get ICDB components for a five-bit up counter.

```
ICDB("command: component_query;
component:counter;
function:(INC);
attribute:(size:5);
ICDBcomponents:?s[]",&counters);
```

To use ICDB in a C program, a user should use the function call ICDB(). The parameters of an ICDB() call are passed in the format as shown below.

ICDB("command description string",variable names);

The command string has the following format:

left hand side(a name):right hand side(a value)

Terms are delimited by a semicolon (;). The left hand side name identifies the attribute name (keyword), and the right hand side specifies the attribute value.

If the right hand side value is supplied by a C variable, it should have a variable description. The variable description has two parts. The first part indicates that the variable is an input to ICDB (denoted by %) or an output from ICDB (denoted by ?). In the above example, **implementation:?s[]** means that the name array (denoted by **?s[]**) of component implementations are to be put into the variable **counters**. This scheme is similar to the scanf() in the C language. Then a user can use these names to communicate with ICDB. In the second query in the above example, **implementation:%s** means that a single string is used as input to ICDB. This scheme is the same as printf() in the C the

language.

The second query is used to get the functions that can be performed by a generated ICDB component. In this ICDB command, ?s[] means that ICDB will output an array of strings to the variable *counters*. The %s construct used below means that a string stored in the C variable *counters*[*i*] is to be input to ICDB.

```
ICDB("command: component_query;
ICDBcomponents:%s;
function:?s[]"
counters[i], &functions);
```

3.2.2. Component request

To get the area and delay information of a component, the component request command should be used. ICDB will then generate components for this command. Then, synthesis tools can use the component instance query to get the information.

A component specification should be supplied to ICDB. There are three types of component specifications: (1) from a given component name or a component implementation name and its attribute values, (2) from a VHDL netlist in which subcomponents are generated by ICDB, and (3) from an IIF description. If the component specification is (1), then ICDB will search in the

generic component library. If it is (2), ICDB provides a translator and expander to transform it into IIF. The second type of specification is used by the partitioner to get the delay and area information for a cluster of ICDB components. The third specification type is typically used for control logic generation. The control logic synthesis tool generates boolean expressions and a list of registers for a design. Then, this control logic can be generated by the ICDB component server.

The optional constraint inputs are delay and geometry. The delay constraints contain clock width, set up time, and delay from an input to an output. The geometry constraints are port positions and aspect ratios.

The following example shows a query for a five bit up counter. ICDB will generate a component according to these specifications.

```
ICDB("command:request_component;
component_name:counter;
attribute:(size:5);
function:(INC);
clock_width:30;
comb_delay:%s;
set_up_time:30;
generated_component:?s",
c_delay,&counter_ins)
```

ICDB allows users to specify constraints of (1)minimum clock width, (2) delay from any input port to a output port under a output load situation, and (3) set up time for input ports. Users can specify the delay from any input port to

output port under a output load situation. In the above example, the minimum clock width is 30 nanoseconds. In addition, the set up time of any input port should be less than 30 nanoseconds. An example of delay constrains is shown as follows.

```
rdelay Q[4] 10
rdelay Q[3] 10
rdelay Q[2] 10
rdelay Q[1] 10
rdelay Q[0] 10
oload Q[4] 10
oload Q[3] 10
oload Q[2] 10
oload Q[1] 10
oload Q[0] 10
```

The expressions **rdelay Q[0] 10** and **oload Q[0] 10** indicate that any delay from any input port to output port Q[0] should be less than 10 nanoseconds under the situation that Q[0] drives a load that equals ten unit transistors. A user can specify the strategy instead of specifying the delay constraints. For example, a user can specify a strategy **fastest**. The ICDB component server will then generate a component with the shortest possible delay.

ICDB will generate a component according to these specifications. The name of this component is put into the variable **counter_ins** by ICDB.

3.3. Component instance query

The component instance query is used to get information about generated components. ICDB provides information for delay, area, shape function, equivalent port, inverted port, and connection information. This information is useful for operator scheduling, resource allocation and binding. For example, the scheduler needs the delay time of components to determine the clock width. A microarchitecture technology mapper needs the connection information of components. For instance, the technology mapper needs to know which input controls the function of an up-down counter. It also needs the control code for the control port that will invoke the up-count operation. The microarchitecture optimizer needs equivalent port, inverted port, and delay information to optimize the design. For example, if the output of a component is connected to a component which requires an active low input, the optimizer can connect the inverted output to the next component to avoid using an inverter.

The synthesis tool can use the following component instance query to retrieve information about the component *counter_ins*. This query retrieves the delay and shape function:

```
ICDB(" command:instance_query;
generated_component:%s;
delay:?s;
shape_function:?s",
counter_ins,&delay_s,&shape_function_s);
```

The ICDB component server will return the delay information in the **delay_s** string variable and the shape function in a **shape_function_s**. The following delay estimation of **counter_ins** is generated by ICDB using the **counter** described in section3 with enable, updown, parallel load attributes. All of the information is stored in the variable **delay_s**.

```
CW 29.0
WD Q[4] 8.5
WD Q[3] 8.5
WD Q[2] 8.5
WD Q[1] 9.7
WD Q[0] 8.7
WD MINMAX 27.3
SD DWUP 26.7
```

The **CW 29.0** means the minimum clock width of the component is 29.0 nanoseconds. **WD Q[3] 8.5** means the delay from clock rising to **Q[3]** port is 8.5 nanoseconds. The expression **SD DWUP 26.7** means minimum setup time for **DWUP** is 26.7 nanoseconds.

The following shape function is generated by the ICDB area estimator which is stored in the **shape_function_s** variable.

```
Alternative=1 width=12000 height=48000
Alternative=2 width=90350 height=61100
Alternative=3 width=72700 height=73500
....
```

This counter can be laid out using a variable number of strips. Thus, it has different aspect ratio. The **Alternative=3** is put into five strips. It has a width

of 72700 micrometers and height 73500 micrometers.

The layout can be generated by the following query.

```
ICDB(" command:request_component;
instance:%s;
alternative:3;
port_position:%s;
CIF_layout:?s",
counter_ins,pin_locs,&counter_layout);
```

An example of port position assignment is shown as follows.

```
CLK left s1.0
D[0] top 10
D[1] top 20
D[2] top 30
D[3] top 40
d[4] top 50
LOAD left s2.0
DWUP left s3.0
MINMAX right s2.0
Q[0] bottom 10
Q[1] bottom 20
Q[2] bottom 30
Q[3] bottom 40
Q[4] bottom 50
```

Each row assigns one port position. The first column indicates the port name. The second column specifies on which side this port is to be put. The third column specify the relative position of the ports. For example D[0], D[1], D[2], D[3], and D[4] are numbered as 10, 20, 30, 40, and 50. This means D[0], D[1], D[2], D[3], and D[4] are to be placed from left to right. Ports with larger number are placed righter. The connection information, VHDL component

description, and VHDL netlist of component **counter_ins** can be retrieved by the following query.

```
ICDB(" command:instance_query;
instance:%s;
VHDL_net_list:?s;
VHDL_head:?s;
connect:?s",
counter_ins,&counter_net_list,&counter_head,&counter_connect);
```

The queries **VHDL_net_list** and **VHDL_head** are for a synthesis tool to build a VHDL netlist of ICDB components to simulate the synthesis result. The **connect** information is used to connect this component. Part of the result of connect information returned by the above query is listed as follows:

```
## function INC
O0 is O0 high
** DWUP 0
** ENA  0
** LOAD 1
** CLK 1 edge_trigger
....
```

This information indicates that to execute the **INC** function the DWUP port should be supplied with 0, ENA port with 0, LOAD port with 1, and the CLK port with a rising edge.

The results of the previous three queries (component query, component request, and component instance query) form an area/time tradeoff graph for the up-counter. This graph is shown in Figure 5.

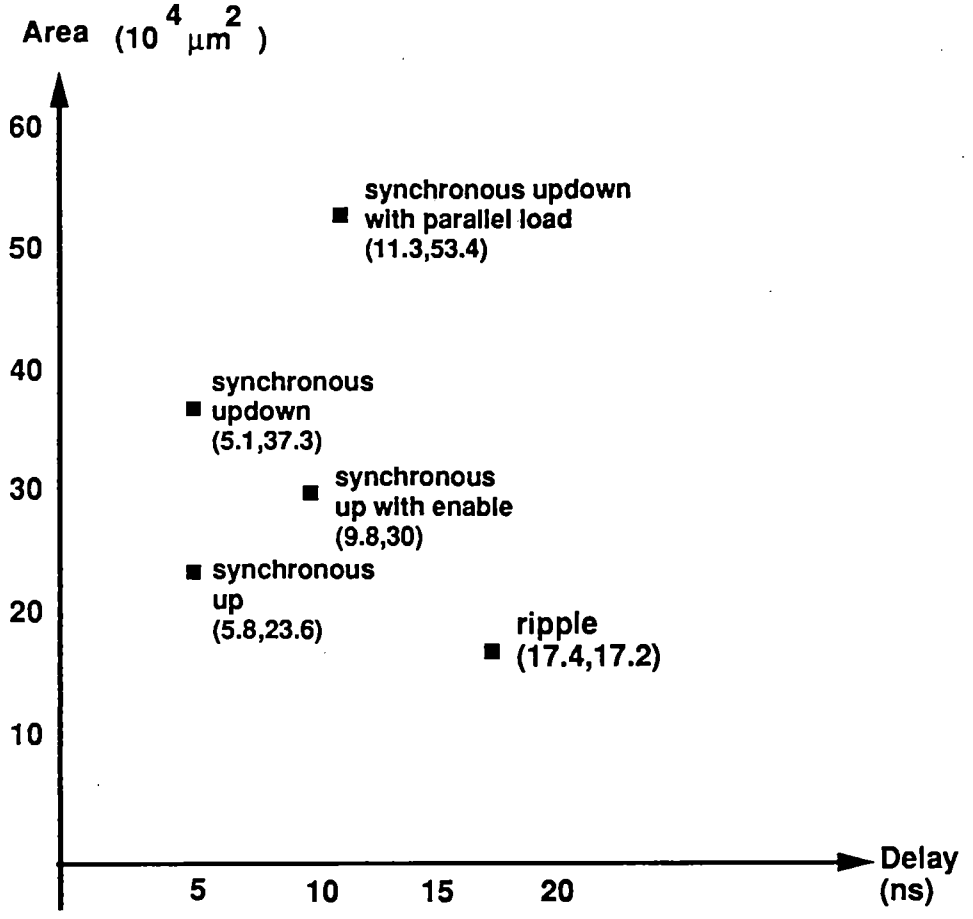


Figure 5 Area/time tradeoff curve of counters

All the components in it can used as an up-counter. A shape function for the updown counter of Figure 5 is shown in Figure 6.

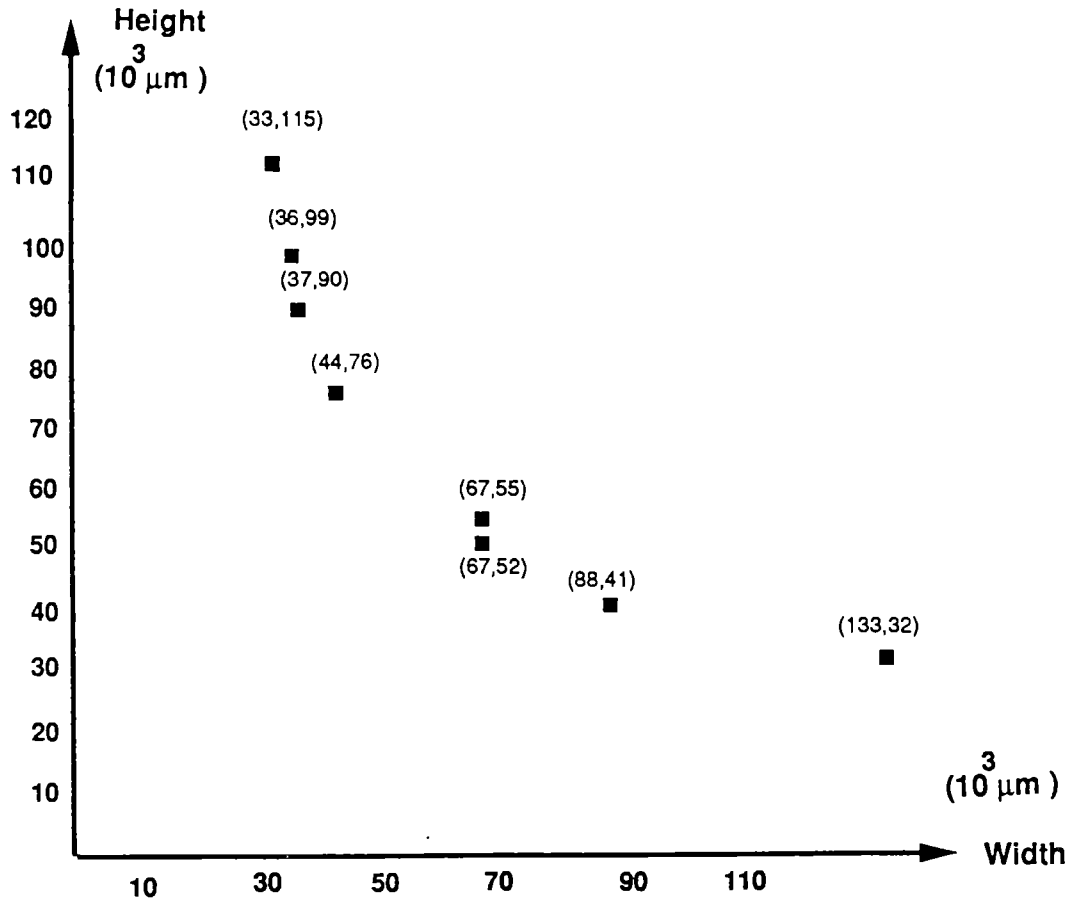


Figure 6 Shape function of the updown counter

4. Component generation and management

ICDB contains two types of components: (1) fixed components and (2) parameterized components. Component generators can generate the layout for components from fixed components or parameterized components when given parameter values. These components are stored in the generic component library.

ICDB has an embedded component generator which takes IIF descriptions and desired constraints and provides delay and shape function estimates, and generates layouts. The constraints include delay, port positions, and shape. Other component generators can be inserted through the knowledge server.

4.1. Component management

ICDB stores its components in the generic component library. An ICDB component consists of two types of descriptions: (1) design data such as IIF descriptions, VHDL netlists, and CIF descriptions, and (2) ICDB data. The ICDB data includes (1) the component type, (2) the function that the component performs, (3) connection information, (4) I/O port descriptions, (5) parameter descriptions, (6) attributes, and (7) file names of the component design data. The component design data is stored in files, while the ICDB data is stored in the INGRES database. When a component is inserted into ICDB, the ICDB data is inserted into the INGRES database.

ICDB components are classified and retrieved by either a component type or the functions they perform. A component type is the name of a microarchitecture component such as counter, register, and adder. A function is an operation that a microarchitecture component may perform such as ADD, and SUBTRACT. For example, an up-counter performs the functions **INCREMENT** and **COUNTER**, a register performs the function **STORAGE**, and

an updown counter with parallel load and enable performs **INCREMENT**, **DECREMENT**, **COUNTER**, and **STORAGE** functions. When a user needs a register, ICDB will search the components which perform the **STORAGE** function. Both the updown counter and the register component will be returned by ICDB. If an optimizer wants to get a component that executes both the **COUNTER** and **STORAGE** functions, the updown counter will be returned. There are only two levels of hierarchy: functions and components. Synthesis tools can request components that execute multiple functions. If there is a component in ICDB which contains the specified functions, ICDB returns it to the synthesis tools. This scheme eliminates the need of synthesis tools to keep track of all kinds of multiple function components.

If a component performs multiple functions, the connection information will be retrieved. This information tells the synthesis tool how to connect a component and how to invoke its functions. An example of connection information for the updown counter is described in the following.

```
## function INC
O0 is O0 high
** DWUP 0
** ENA  0
** LOAD 1
** CLK 1 edge_trigger
```

This information indicates that to execute the **INC** function the DWUP port should be supplied with 0, ENA port with 0, LOAD port with 1, and the CLK

port with a rising edge. Thus, synthesis tools know how to use a component.

4.2. Tool management

Tool programs in ICDB are formed into a set of component generators. Each component generator should be able to generate component layouts from a given description language and constraints, such as IIF and constraints, and to produce the delay and shape functions of generated components.

The component implementations described by a description language are stored in a library of the ICDB generic component library. When a user requests a component, ICDB will query the generic component library to get available component implementations. ICDB checks the format type of the design data and invokes a component generator to generate the requested component. From the user's point of view, ICDB stores many components that can achieve different requirements and constraints. Both the fixed components and components generated by component generators have the same user interface. More component generators make ICDB become a larger component library. This scheme is shown in Figure 7.

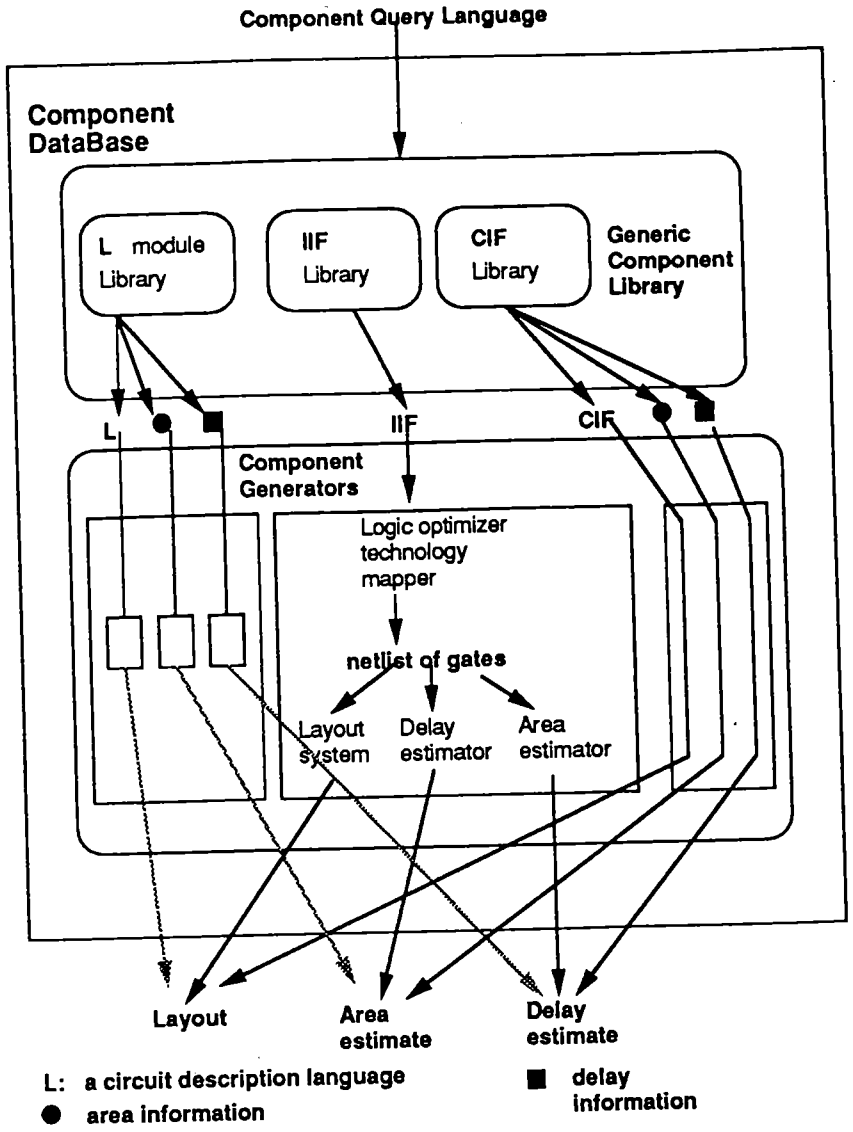


Figure 7 ICDB organization

A tool which does not belong to any component generator will never be used by ICDB. A tool can be used by more than one component generator.

A component generator has two steps. The first step takes a design data description and produces delay and shape function estimates. The second step executes from the end of step 1 and generates the layout. A component generator is defined by a list of tuples: (step-no, tool-name). It is executed in a straight sequence.

Each tool to be inserted into ICDB should be accessible via a shellscipt. The parameter descriptions for each tool are stored in INGRES. Those parameters describe the type of the tool's input and output data. ICDB defines a set of standard formats such as formats for delay constraints, port position constraints, delay descriptions and shape function descriptions. The translation programs between ICDB's format and the tool's format are embedded in the shellscript. Thus, ICDB is able to pass data between component generators and synthesis tools without the need for a data translation program.

When a component generator is invoked, a series of shellscripts are executed. All the tools retrieve data from ICDB and place the resulting data back into ICDB. ICDB retrieves data from synthesis tools and then returns the data to the synthesis tools.

4.3. Component generation

The ICDB embedded component generator includes (1) an IIF expander, (2) a logic optimizer and technology mapper, (3) a transistor sizing program, (4)

a layout generator, (5) a timing estimator, and (6) an area estimator. In addition, a VHDL simulator and a circuit simulator are provided to verify the correctness of functionality and whether the timing constraints are met. The component generation path is shown in Figure 8.

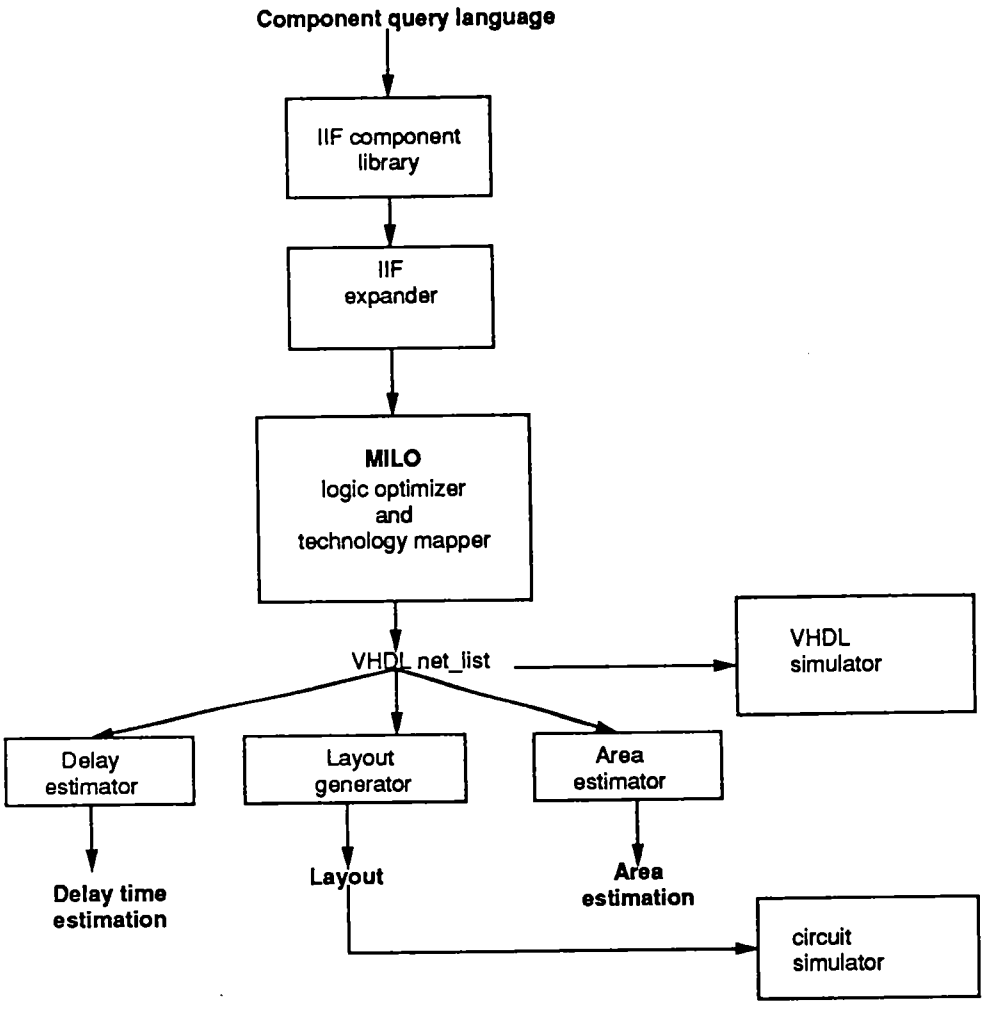


Figure 8 ICDB component generation path

4.3.1. Logic synthesis and technology mapping

The logic synthesis and technology mapping tool accepts expanded IIF and delay constraints and produces a VHDL netlist. It consists of 6 steps. First, the sequential constructs are removed, creating a set of boolean equations. These equations are minimized and then factored by a logic optimizer. In the second phase, it reduces the number of levels along the longest paths by performing factoring. A third step then performs technology mapping by combining gates into complex gates. Sequential logic is then reinserted. The fourth phase sizes the transistors according to the input delay constraints. A VHDL netlist of gates with assigned transistor sizes is produced and can be passed to a custom layout generator.

4.3.2. Layout generator

ICDB has a set of layout tools that can generate layouts from a netlist. The netlist consists of gates, complex gates, and flip-flops, for which different transistor sizes can be given. The layout tools uses a two dimensional layout in which components can be placed into a number of layout strips. Each strip has a pair of Vdd/Vss lines setting its boundaries. Two neighboring strips share a common Vdd/Vss line. Users can assign the number of strips to be laid out and the I/O port positions of a component.

4.4. Delay and shape function estimation

ICDB can generate the gate-level netlist for most microarchitecture components under five minutes. Then ICDB uses a delay estimator and an area estimator to estimate the component's delay and shape function.

4.4.1. Delay estimation

ICDB stores three types of delay information for each basic cell: (1) the delay increase for each additional unit of transistor load, (2) the delay from an input port to an output port, and (3) the delay increase for each additional fanout. Suppose an output port drives **Trans_no** unit transistors and is connected to **fanout_no** input ports. If the delay information is X for (1), Y for (2), and Z for (3), the delay of an output is estimated by the following formula.

delay=Trans_no * X + Y + fanout_no *Z

The delay of a component is the sum of all the estimated delays of cells along the path.

4.4.2. Area estimation

Two properties of a basic cell are used in the components' area estimation: (1) the cell's width and (2) the number of tracks used. To estimate the strip width, two variables are used: X and Y. X is the maximum width of the strips obtained by placing the cells randomly in each strip so that each strip has the

same number of cells. Y is found by examining different placements of cells in each strip. Different cell placements result in different widths for the layout. One of these placement achieves the smallest width. Y is the width of this placement. The width of a component is estimated by **(X+Y)/2**.

Estimation of the component height is based on (1) transistor height, and (2) the number of tracks. The height of transistors is estimated by the average height of all the transistors. The number of tracks needed is estimated by analyzing the netlist. We estimate the horizontal length required of each net according to its connectivity. The estimated number of tracks used is estimated by total horizontal wire length divided by a track utilization constant. The track utilization function is obtained from experiments on ICDB's layout tool. The function will return a track utilization constant when given the number of cells in a strip.

5. Examples and results

ICDB is written in C language and runs on Sun workstations under the UNIX operating system. We have run a number of examples to demonstrate some of the capabilities of the ICDB. The examples are based on the IIF description for the counter described in section 3.

Figure 5 shows the time/area tradeoff graph for different implementations of the 5 bit up-counter. The Y axis of the graph represents the area of the

components and the X axis is the delay to output port Q[4]. The counters were generated by providing different parameter values. Layouts of these components are shown in Figure 9.

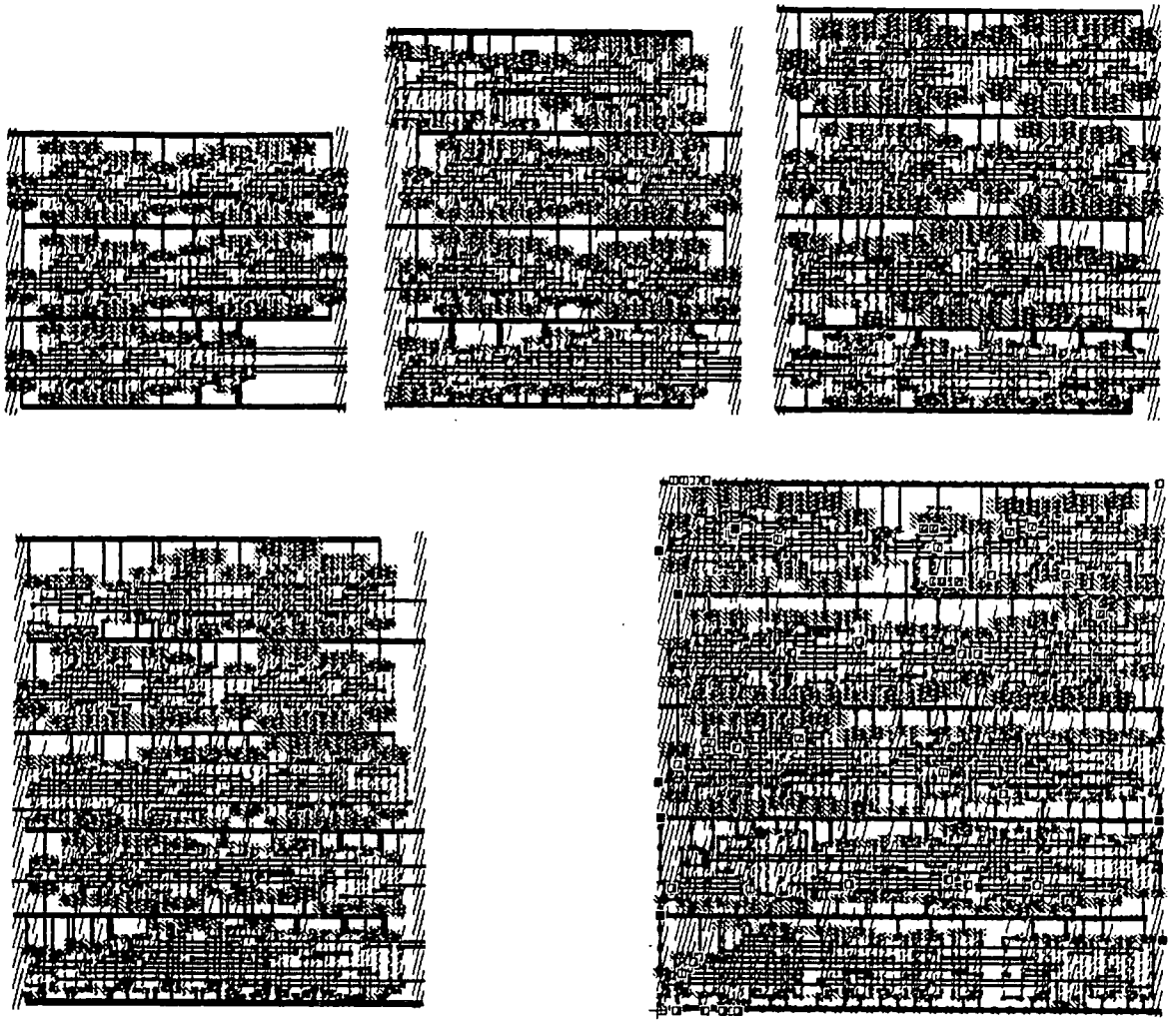


Figure 9 Layouts of counters

In further refinements of a design, the component output loads and delay constraints may be changed as well.

Figure 10 shows the area/output-load tradeoff curve for the synchronous updown counter of Figure 5.

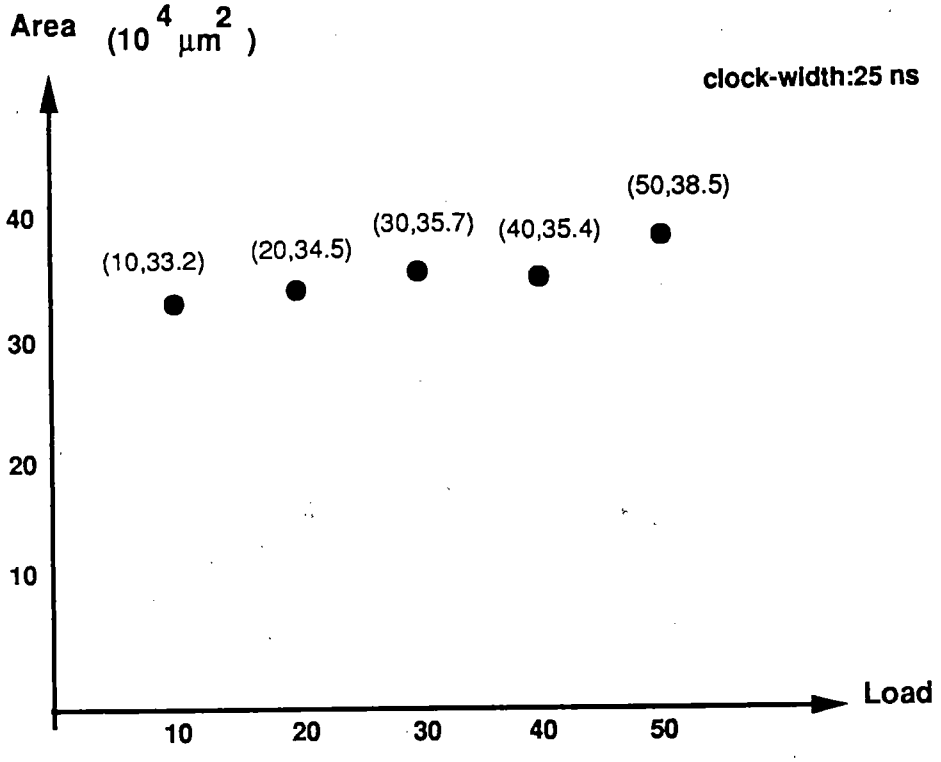


Figure 10 Area/load tradeoff curve of the updown counter

The required minimum clock width of this counter was set at 25 nanoseconds. Then different output load requirements were requested - ranging from 10 to 50. ICDB sized the transistors of the counter to make it achieve the same minimum clock width. The results show that the area increased only 6 percent when the output load constraint was changed from 10 to 40.

The area/clock-width tradeoff curve of the counter is shown in Figure 11.

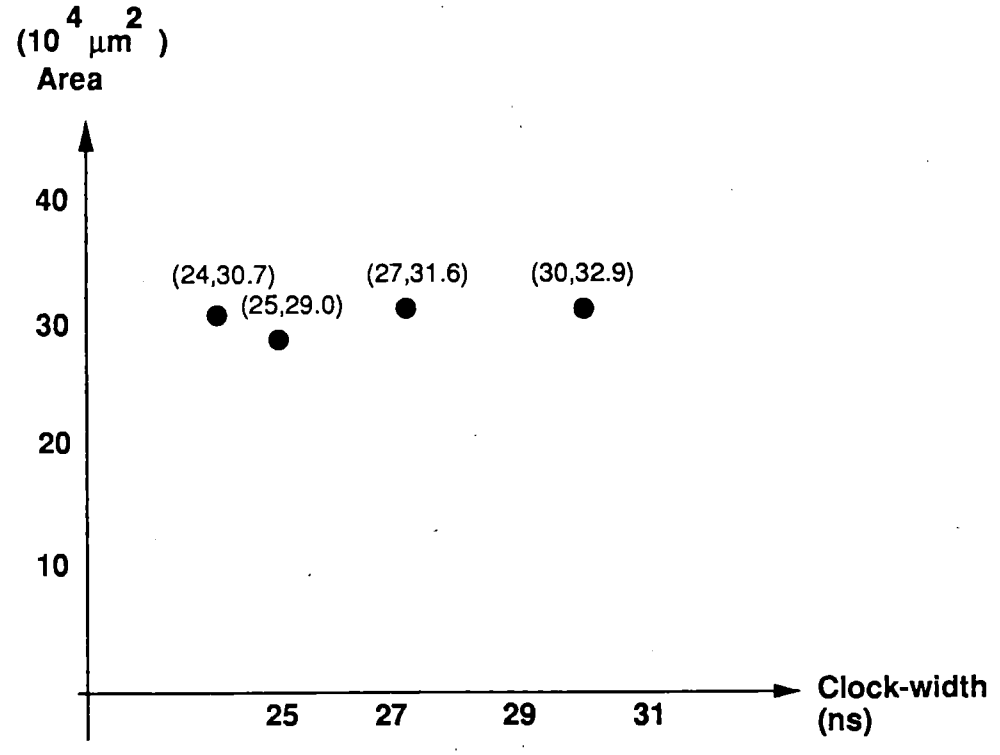


Figure 11 Area/Clock-width tradeoff curve of the updown counter

In this figure, the output loads of the counter were held constant at 10 units. The minimum clock width was varied from 24 nanoseconds to 30 nanoseconds. Figure 11 shows that the area range is within 6 percent. Tightening the delay constraints does not increase the area in all cases. ICDB will generate many components from the same netlist structure with different transistor sizes. With different transistor sizes, the layout system will perform a different placement and routing. Thus, it may produce a larger layout even though the total transistor size is smaller.

Different aspect ratios of a component are provided by laying it out with a different numbers of strips. Figure 6 shows the shape function for the updown counter. Layouts using different aspect ratios from the shape function are shown in Figure 12.

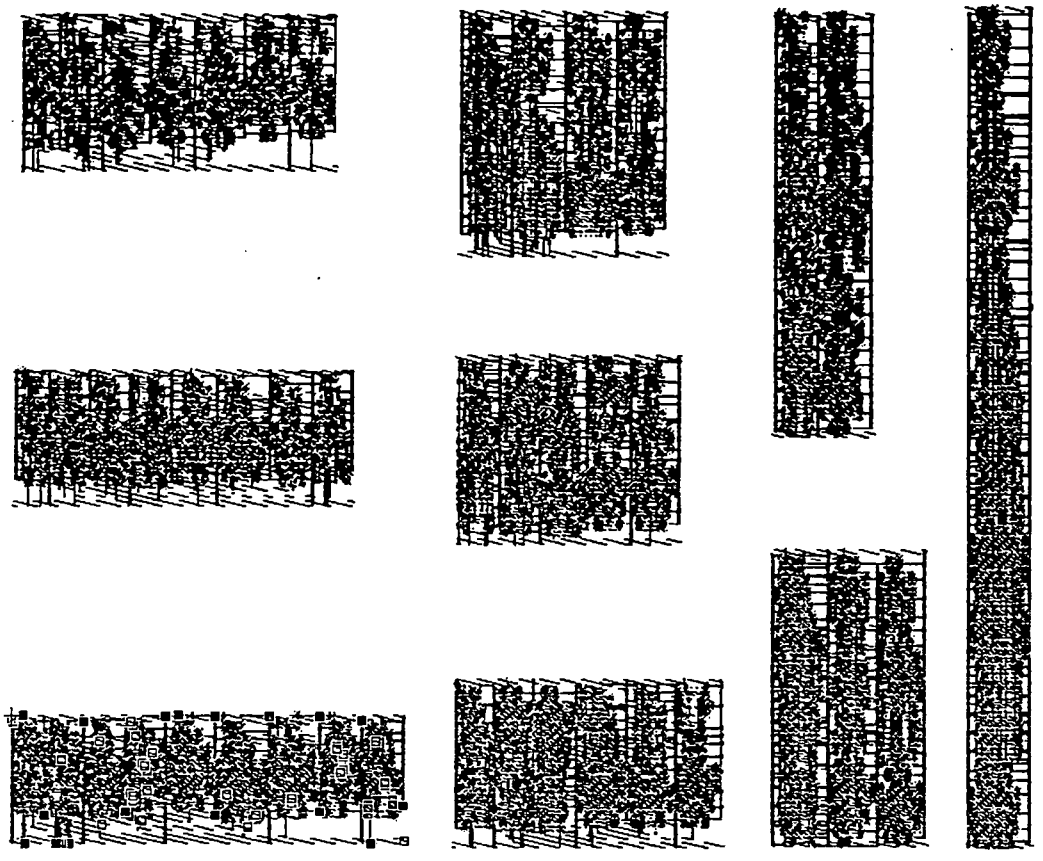


Figure 12 Different shape layouts of the updown counter

The port positions can be assigned when a user requests the generation of the layout.

Figure 13 shows two layouts of a simple CPU. The layouts were put together using ICDB generated control logic components. The difference between the two layouts is the position of the control logic component (placed on the left-hand side of the first layout and placed on the bottom of the second layout). For the layout on the left hand side of the figure, the control-logic component was generated in a tall and thin shape. Ports were placed at the left side of the layout to form a one to one aspect ratio for the layout. In the layout on the right side, the control logic component was laid out short and wide, and the ports placed on the both sides. This produced a 2 to 1 aspect ratio layout.

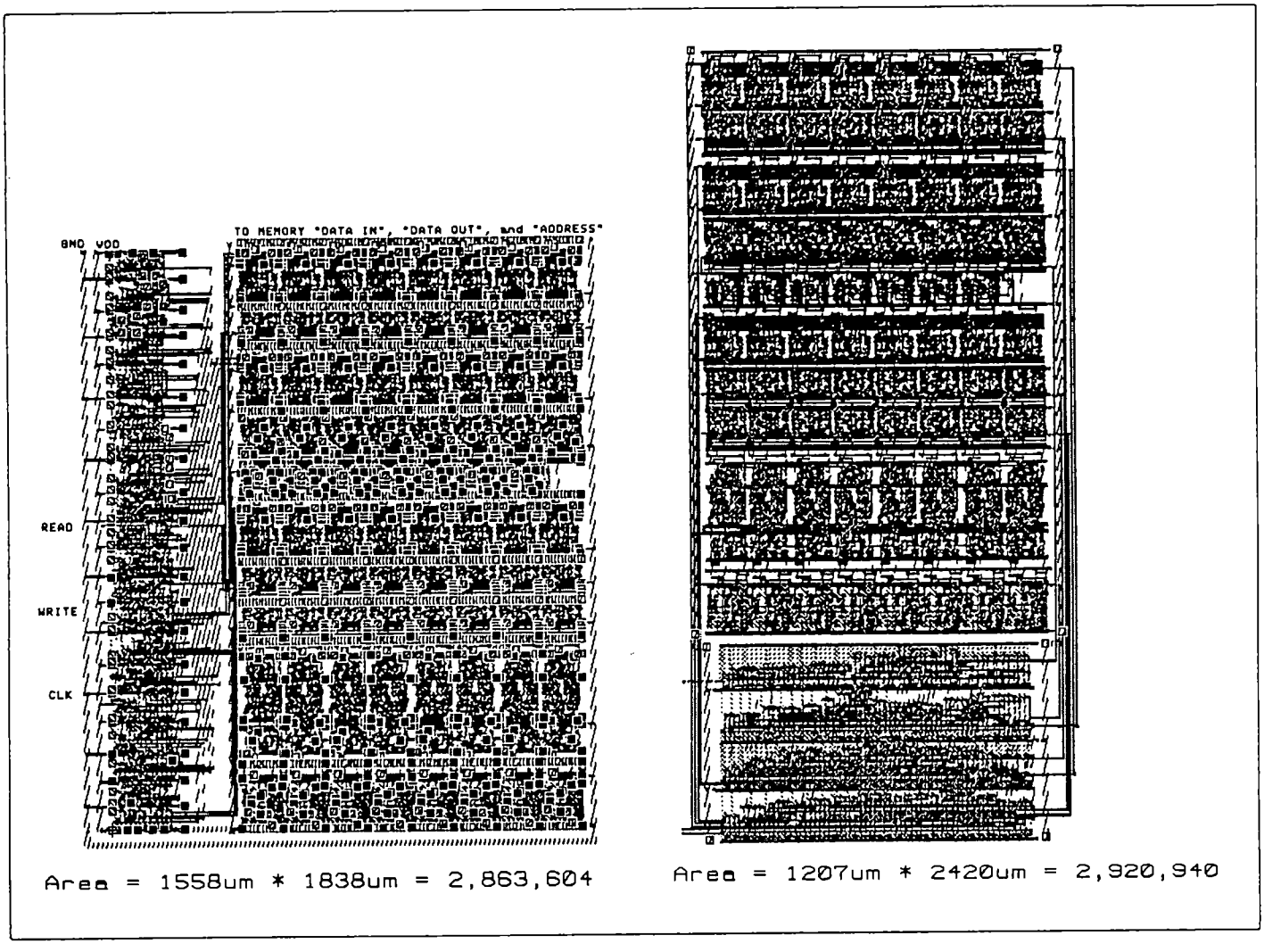


Figure 13 The layout of a simple computer

6. Conclusion

We have described an Intelligent Component DataBase system which delivers components for synthesis tools when given a set of attributes and constraints. It can provide logic level descriptions for gate arrays or standard cell methodologies or layouts for custom methodologies. In this paper we also introduced the concept of a tool-transparent component server for high-level synthesis, so that synthesis tools do not have to deal with tradeoffs below the component level. We developed a language IIF for storing parameterizable components as well as a new query language for use by high-level synthesis tools. Additional contributions are a new management scheme for components, a set of generators and tools, the definition of exploratory strategies for high-level synthesis, and the development of estimators for the logic and layout levels. However, the main contribution of this paper is the demonstration that component libraries and component catalogs with hundreds and hundreds of pages are no longer necessary.

Acknowledgements

We would like to thank Allen Wu for preparing Figures 9,12,13. The unparameterized IIF was defined by Nels Vander Zanden. We also want to thank Nels for his helpful comments during the writing of this paper.

Gwo-Dong Chen is thankful for the financial support provided by the government of Repulic of China.

REFERENCE

[BeOw87] J. A. Beekman, R. M. Owens, M. J. Irwin, "LOGICIAN: A tool for their Efficient Generation", Proc. of 24th Design Automation Conference, pp. 357-362.

[BrRu87] R. Brayton, R. Rudell, Sangiovanni-Vincentelli, and A. Wang "MIS: A Multiple -Level Logic Optimization System", IEEE Trans. on CAD, Vol. CAD-6, no.6, Nov. 1987.

[ChCh89] C. C. Chen, S. L. Chow, "The Layout Synthesizer: An Automatic Netlist-to-Layout System" Proc. of 26th Design Automation Conference, pp. 232-238.

[ChMa88] Edmund K. Cheng, Stanley Mazor, "The Genesil Silicon Compiler" in Daniel D. Gajski ed. Silicon Compilation pp.361-405, AddisonWesley 1988.

[CoSn88] Vince Corbin, Warren Snapp, "Design Methodology of the Concorde Silicon Compiler" in Daniel D. Gajski ed. Silicon Compilation pp.406-455, AddisonWesley 1988.

[Dutt88] Nikil D. Dutt, "GENUS: A Generic Component Library for high level synthesis" Technical Report 88-22, Department of Information and Computer Science, University of California Irvine.

[FiDu85] J. P. Fishburn and A. E. Dunlop, "TILOS: a Posynomial Programming Approach to Transistor Sizing" Digest of Technical paper ICCAD-85.

[He87] Hedlund, K. Kye, "Aesop: A Tool for Automated Transistor Sizing" Proc. of 24th Design Automation Conference, pp. 114-119.

[Ke87] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching" Proc. of 24th Design Automation Conference, pp. 341-346.

[LiGr87] Youn-Long Lin, Daniel Gajski, "LES: A Layout Expert System" IEEE trans. on CAD, Vol. CAD-7, no. 8, pp. 868-876, August 1988.

[OnLL89] C. L. Ong, J. T. Li, C. Y. Lo, "GENAC: An Automated Cell Synthesis Tool", Proc. of 26th Design Automation Conference, pp. 239-245.

[RDVG88] J. Rabaey, H. De Man, J. Vanboof, G. Goosens, and R. H. J. M. Otten, "CATHEDRAL II: A Synthesis System for Multiprocessor DSP Systems" in Daniel D. Gajski ed. Silicon Compilation pp.311-360, AddisonWesley 1988.

[McFa86] Michael C. McFarland, S.J., "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions" Proc. of 23th Design Automation Conference, pp. 474-480.

[McFa87] Michael C. McFarland, S.J., "Reevaluating the Design Space for Register-Transfer Hardware Synthesis" Digest of Technical paper ICCAD-87, pp.262-265.

[TrDi89] Michael T. Trick and Stephen W. Director, "LASSIE: Structure to Layout for Behavioral Synthesis Tools" Proc. of 26th Design Automation Conference pp.104-109.

[JKMP89] Rajiv Jain, Kayhan Kucukcakar, Mithchell J. Mlinar, and Alice C. Parker, "Experience with the ADAM Synthesis System" Proc. of 26th Design Automation Conference pp. 56-61.

[VaGa88] Nels Vander Zanden, Daniel Gajski, "MILO: A Microarchitecture and logic optimizer" Proc. of 25th Design Automation Conference pp. 403-408.

[VaWG89] Nels Vander Zanden, Allen Wu, Daniel Gajski, "Performance Optimization in Layout Driven Synthesis", Technical Report 89-21, Department of Information and Computer Science, University of California Irvine.

[ThDW88] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nester, R. L. Blackburn, "The System Architect's Workbench" Proc. of 25th Design Automation Conference pp.337-343.

[Wayn86] Wayne Wolf, "An Object Oriented Procedural Database for VLSI Chip Planning", Proc. of 23th Design Automation

APPENDIX A

IIF description

(This appendix was written by Nels Vander Zanden and Gwo-Dong Chen.)

1. Irvine Intermediate Form (IIF)

In order to describe micro_architecture_level components (such as counter, shift registers, etc.), we need a format capable of describing sequential, asynchronous behavior, and I/O conversion. In this document, we define the Irvine Intermediate Form (IIF) which extends a boolean expression language with clocking and asynchronous behavior in order to describe generic components composed of logic gates, flip-flops with asynchronous set and reset, and interface components. In addition, IIF provides programming structures for describing parameterized objects. The programming structures include IF, FOR loop, and function call which enable designers to specify a component with replicatable structure.

A macro expander will translate the IIF description to a nonparameterized IIF description. A logic optimizer and technology mapper MILO [VaGa 89] can accept a nonparameterized IIF description, and generate a net_list using logic gates and complex cells.

IIF is an extension of the Berkeley EQN (equation) format for describing boolean expressions. Besides providing the basic boolean operations of AND, OR, NOT, XOR, and XNOR, IIF contains operators for specifying a D flip-flop with asynchronous set and reset and operators for tri-state, delay, schmitt trigger, and wire-or. Any component in the GENUS library [Dutt 88] is representable in IIF. An IIF file has the following format:

```
design name
signal and variable declarations

/* This is a comment. */
{

list of equations

}
```

An IIF description is divided into 2 parts. The first part describes signals and variables description. The second part is the design description. All the signals and variables must be declared before they are used in the second part. A signal of IIF is a net in a design, while a variable is used for design parameters. Comments can be placed anywhere between /* and */ . Each IIF statement is delimited by ;.

The design description section is a list of IIF expressions enclosed by { and }. IIF has a block structure language similar to the C programming language. A

block structure in IIF can be viewed as a hardware block. The block is described by a list of IIF expression as described in section 1.2. A designer can use a parameterized structure, described in sec 1.3, to assemble small blocks into a component.

2. IIF declarations

An IIF declaration consists of the following elements.

NAME: name of design;
PARAMETER: variable list;
VARIABLE: variable list;
INORDER: input signal list;
OUTORDER: output signal list;
PIIFVARIABLE: internal signal list;
SUBFUNCTION: IIF subfunction list;
SUBCOMPONENT: subcomponent used;

The declaration part of IIF specifies the name of the design, variables and signals used in the description part of an IIF description. The keyword **NAME** is used to assign the design name. The description after keywords **INORDER**, **OUTORDER**, and **PIIFVARIABLE** specify signals. The variables are declared after keywords **PARAMETER** and **VARIABLE**.

A signal in IIF is the same as a boolean variable in a boolean expression or the name of a net in a net list. A component has three kinds of signals: (1) input, (2) output, and (3) internal. Input signals are declared in the **INORDER** signal list and output signals in the **OUTORDER** signal list. The internal

signals of a component are specified in the **PIIFVARIABLE** signal list.

A variable in IIF is used for parameterized component description. There are two types of variables. One is the parameter to be given by users according to their requirement. For example, in an n_bit adder description, the bit length is a parameter. The second variable type is the same as a variable in C programming language. It is used in parametrized structure description of IIF. The keyword **PARAMETER** is used to specify the parameter variables of this design. The keyword **VARIABLE** is used to declare the variables required in describing this design.

A signal or a variable can be simple one or indexed. The indexed variable declaration in IIF is the same as in C. The variable description **Var[3]** means Var[0], Var[1], Var[2]. It is suggested that a signal name begin with a capital letter.

3. IIF expression

An IIF expression is a boolean equation with some new operators. The followings are operators of IIF.

Binary operators:

+	OR
*	AND
(+)	EXOR
(.)	ENOR

=	assignment
@	AT (for specify D flip flop clock)
~a	asynchronous assignment for D flip flop
/	asynchronous AT
~d	DELAY
~t	TRISTATE
~w	WIRE OR

Unary operators:

!	NOT
~b	BUFFER
~s	SCHIMITT TRIGGER
~f	falling edge trigger clocking (for flip-flop)
~r	rising edge trigger clocking (for flip-flop)
~h	level clocking active at high (for latch)
~l	level clocking active at low (for latch)

The +, *, !, (+), and (.) operators are used in Boolean expression. IIF extends boolean expressions with operators for expressing D flip-flops with asynchronous set and reset (@, ~a), buffers (~b), schmitt triggers (~s), delay elements (~d), tristate buffers (~t), and wire ors (~w). The ~f, ~r, ~h, and ~l operators are used to specify the clocking of the D flip flop.

Sequential logic is represented in the form:

Var = (boolean input equation)
@ (clock_operator clock_expression)
~a (asynchronous_set_reset_expression)

For example, a falling edge triggered with set and reset is described as follows:

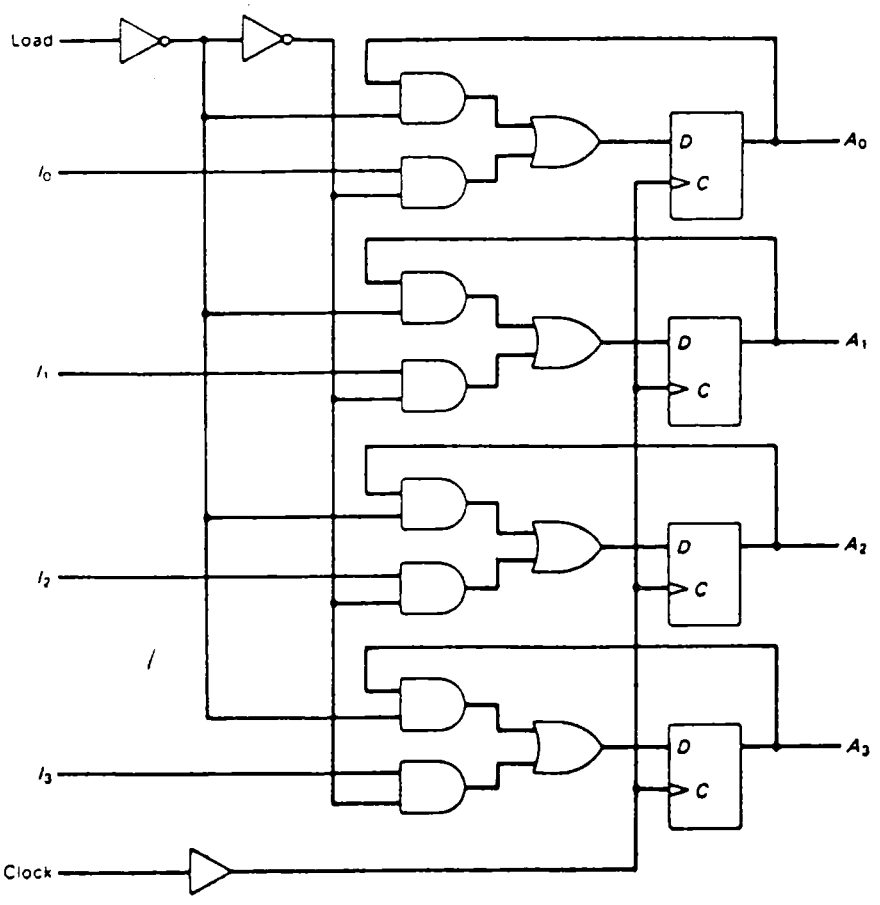
Q=(D @ ~f clk) ~a (0/!reset,1/!set)

The expression **Q = D @ ~f clk** means that **Q** is set to D at the falling edge (denoted by **~f**) of the signal **clk**. The expression **~a (0/!reset,1/!set)** indicates that **Q** is set to 0 when reset signal is 0 (denoted by **0/!reset**) and is set to 1 when set signal is 0 (denoted by **1/!set**). The asynchronous expression is a list of descriptions in the form as **value/condition**. It means that when the **condition** expression is true, the output is set to **value**.

The ~b and ~s operators are similar to ! (Not). !A represents the output signal of an inverter with input signal A. Likewise, ~b X is output signal of the buffer with X as its input signal and ~s Y of the schmitt trigger with input Y.

Both ~d and ~t are binary operators and the position of operands is significant. **A ~d 10** indicates the output signal of a delay element which has a 10 ns delay from input to output. The expression **Cl ~t control** indicates the output signal of a tri-state buffer, of which **Cl** is the input and **control** is the control signal. A wired or signal of **A** and **B** is expressed as **A ~w B**.

Example 1: Register with parallel load




```
INORDER = Load,I0,I1,I2,I3,Clock;
OUTORDER = A0,A1,A2,A3;
PIIFVARIABLE = not_load,load,CP;
{
CP = ~b Clock;
not_load = !Load;
load = !not_load;
A0 = ((I0*load) + (A0*not_load)) @ (~r CP);
A1 = ((I1*load) + (A1*not_load)) @ (~r CP);
A2 = ((I2*load) + (A2*not_load)) @ (~r CP);
A3 = ((I3*load) + (A3*not_load)) @ (~r CP);
}
```

4. Parameterized structure

In order to describe components with different bit length, IIF must be extended with additional language constructs.

In designing a 16 bit adder, we may start with a one bit adder, and then replicate it sixteen times to built the 16 bit adder. Sometimes, we use some existing components to build other components. For instance, we use an adder and exclusive OR gates to build an adder_subtractor. Therefore, IIF provides facilities for using replicated structure to describe a parameterized component.

IIF has the same block structure and the similar syntax as C language. There are three programming constructs in IIF, sequence: decision, and loop. The sequence structure is a list of statements enclosed by { and }. The **IF** statement is used for decision and the **FOR** statement for looping. Programming language theory proves that any structure can be described with three basic structures: sequence, decision, and loop. So IIF is complete from the structure point of view.

4.1. C expression

IIF provides C expressions for IIF users so they can write an IIF program as a programming language. C expressions might appear in the index description of a signal declaration. For instance, if a decoder has an n_bit input, then it has 2**n_bit outputs. The OUTORDER description can be specified as

OUTORDER: O[2**n];.

It can be used in description part of IIF, for the index of a indexed signal and in IIF parameterized description.

Operators allowed in C expressions are +, -, *, /, ** (exponential), =, ++, and --. The syntax of C expression is the same as C language.

A designer can write a C expression statement using an IIF **#c_line** statement. For example, consider a program that calculates C(n,m)= (n!)/((n-m)!*(m!)). It can be written using a language construct and the #c_line statement.

```
#c_line cnm = 1;
#for(i=1;i<=m;i++)
#c_line cnm = cnm * (n-i-1) * (i);
```

4.2. Replicated structure in IIF

A **for** loop structure is used to facilitate replication of a basic building unit. The syntax is the same as C. The **","** operator is not allowed in IIF statements. For example, the following statement is not allowed in IIF:

```
#for( i=0,j=0; i<size; i++,j++ )
```

Take an adder as an example. We use the indexed signals **I0, I1, O and C** ,and the for loop **for(j=0 ;j< nib; j++)** to describe the adder.

Example 2: An n bit ripple carry adder described by IIF.

```
NAME: ADDER;
PARAMETER: size;
INORDER: I0[size], I1[size], Cin;
OUTORDER: O[size], Cout;
PIIFVARIABLE: C[size+1];
VARIABLE: i;
{
  C[0]=Cin;
  #for(i=0;i<size;i++)
  {
    O[i] = I0[i] (+) I1[i] (+) C[i];
    C[i+1] = I0[i]*I1[i] + I0[i]*C[i] + I1[i]*C[i];
  }
  Cout = C[size];
}
```

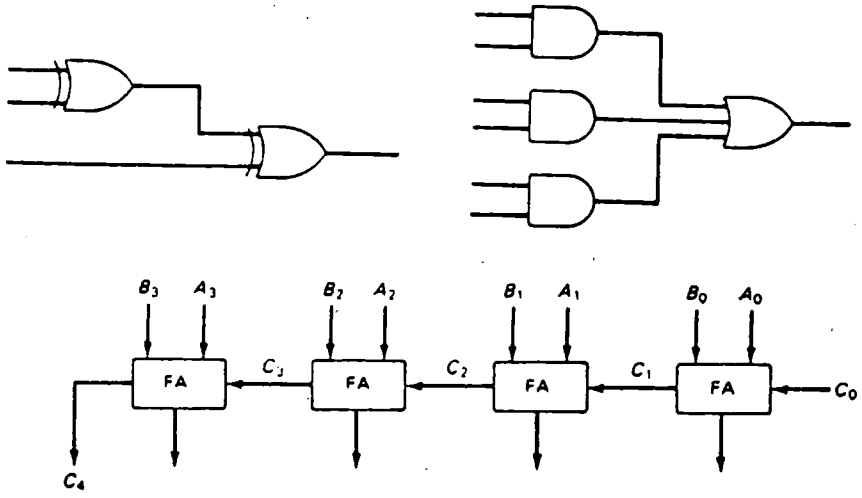


Figure-1 Schematic of an adder

The component name of this description is ADDER. It can execute the function ADD. Figure-1 shows a schematic for this program. The statements in the for loop describe a one bit counter as in Figure 1(a). Figure 1(b) depicts the

building structure described by the for loop. A 4 bit adder can be generated by supplying a file with the following content to the IIF expander.

adder1 4 A B 0 S Cin COUT C

The parameter file should contain a value for each signal/variable in IIF declaration part with the same order as they appeared in IIF.

The following IIF description is generated from the above IIF program by supplying the above parameter values. This format is used as the input file of the MILO logic optimizer and technology mapper. Note that the EXOR operator is represented by !=.

```
NAME=adder1;
INORDER= CIN A[0] A[1] A[2] A[3] B[0] B[1] B[2] B[3];
OUTORDER=COUT O[0] O[1] O[2] O[3];
    CIN=0;
    C[0]=CIN;
    S[0]=A[0]!=B[0]!=C[0];
    C[1]=A[0]*B[0]+C[0]*A[0]+C[0]*B[0];
    S[1]=A[1]!=B[1]!=C[1];
    C[2]=B[1]*B[1]+C[1]*B[1]+C[1]*B[1];
    S[2]=A[2]!=B[2]!=C[2];
    C[3]=B[2]*B[2]+C[2]*B[2]+C[2]*B[2];
    S[3]=A[3]!=B[3]!=C[3];
    C[4]=A[3]*B[3]+C[3]*A[3]+C[3]*B[3];
    COUT=C[4];
```

4.3. IIF Function

In many cases, we will use a predefined function to define a new one. Thus, IIF provides a function facility to do this. The parameter passing method in IIF

is call_by_name as a macro line function defined by **#define** in C language as follows:

#define max(A,B) ((A) > (B) ? (A) : (B))

For example, if we want to develop an adder_subtractor, we can use the ADDER function defined before.

Example 3: an adder_substrator is defined by calling the adder component described in example 2.

```
NAME: ADDSUB;
PARAMETER: size;
INORDER: A[size],B[size],ADDSUB;
OUTORDER: O[size],Cout;
PIIFVARIABLE: C[size+1],B1[size];
VARIABLE: i;
SUBFUNCTION: ADDER;
{
      #for(i=0;i<=size;i++)
       { B1[i]=ADDSUB(+)B[i]; }
       #ADDER(size,A,B1,ADDSUB,O,Cout,C);
}
```

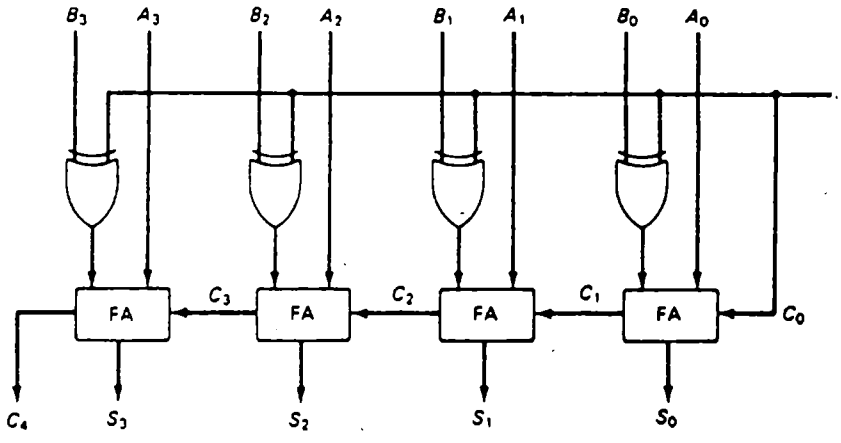


Figure-2 Schematic of a 4-bit adder_subtractor

If we call the IIF expander using this IIF description and the following parameter values (adder_subtractor4 4 ADDSUB A B COUT O), then IIF expander will expand it into the following statements. Figure-2 shows the schematic of it.

```
NAME:adder_subtractor4
INORDER:A[0] A[1] A[2] A[3] B[0] B[1] B[2] B[3] ADDSUB;
OUTORDER:O[0] O[1] O[2] O[3] COUT;
B1[0]=ADDSUB!=B[0];
B1[1]=ADDSUB!=B[1];
B1[2]=ADDSUB!=B[2];
B1[3]=ADDSUB!=B[3];
C[0]=ADDSUB;
O[0]=A[0]!=B1[0]!=C[0];
C[1]=A[0]*B1[0]+C[0]*A[0]+C[0]*B1[0];
O[1]=A[1]!=B1[1]!=C[1];
C[2]=B1[1]*B1[1]+C[1]*B1[1]+C[1]*B1[1];
O[2]=B1[2]!=B1[2]!=C[2];
C[3]=B1[2]*B1[2]+C[2]*B1[2]+C[2]*B1[2];
O[3]=B1[3]!=B1[3]!=C[3];
C[4]=A[3]*B1[3]+C[3]*A[3]+C[3]*B1[3];
COUT=C[4];
```

4.4. IF statement

Sometimes in a replicated structure we have different input connections for certain subfunctions. Thus, IIF provides an IF statement to deal with this condition. For example, the IF statement can be used in defining a shifter as shown below.

Example 4: A left shifter with shift_distance control and 0 filled in on the right.

```
NAME:SHL0
FUNCTIONS:SHL0
PARAMETER:size shift_distance
INORDER:I[size]
OUTORDER:O[size]
{
 #for(i=0;i<size;i++)
   {
   #if(i<= shift_distance -1 )
      O[i]=0;
   #else
      O[i]=I[i-shift_distance];
   }
}
```

4.5. Aggregate structure

Sometimes we need to describe a device in which the number of input pin is a parameter. In IIF, we furnish aggregate assignment operators to describe this type of component. IIF has the following aggregate assignment operators:

+=	aggregate by + OR
*=	aggregate by * AND
(.)=	aggregate by (.) ENOR
(+)=	aggregate by (+) EXOR

Notice that only one statement is allowed inside the for loop and only one level is allowed. For example, an AND function can be defined using this operator.

Example 5: An AND gate with variable number of inputs.

```
NAME: AND;
PARAMETER: size;
INORDER: I0[size];
```

```
OUTORDER: O;
VARIABLE: i;
{
  #for(i=0;i<size;i++)
   O*=I0[i];
}
```

In this example, when size has value 4, the for loop will generate a statement as follows.

O=I0[0]*I0[1]*I0[2]*I0[3]

We can see from this example that the O signal is assigned by a set of signals I0[0], I0[1], I0[2], and I0[3]. The signals are aggregated by the operator *.

APPENDIX A.1: IIF usage

The IIF compiler has 2 phases. The first phase contains a parser and an internal form generator. The second phase use a IIF expander. Both programs, piif1_c and piif3, can be found at

/ch/ub/chen/program/piif/compiler.

The execution sequence is list as follows.

piif1 -i input_file -o internal_form_file -d /dev/null

input-file: filename of IIF

internal_form_file: filename for storing IIF internal form. It will be used in the second phase.

piif2 -i internal_form_file -o file_4_MILO -p parameter_file -d /dev/null

internal_form_file: phase 1 output file storing IIF internal form.

file_4_MILO: file name to store a non parameterized IIF which is used as input to MILO.

parameter_file: file name of parameter value for the component.

APPENDIX A.2: IIF syntax

```
%start prog

%token NAME PARAMETER INORDER OUTORDER VARIABLE SUBCOMPONENT
%token SUBFUNCTION COLON NEW_IDENTIFIER
%token IDENTIFIER INTEGER BUFFER SCHIMIT RISE  FALL HIGHVALUE
%token LOWVALUE DELAY TRISTATE WIREOR AT ASYNC DIVIDE AND OR MOD
%token MINUS LESS GREATER LEQ GEQ EQ NEQ LAND LOR ASSIGN
%token INS_ADD INS_MUL INS_XOR INS_XNOR
%token INC DEC LPAREN RPAREN LBRACKET RBRACKET COMMA IF
%token  ELSE FOR BREAK CONTINUE CLINE SEMICOL NOT XOR XNOR
%token LEFT_CURY_BRACKET RIGHT_CURY_BRACKET
%token NO_SIGN PIIF_VARIABLE_TOKEN
%token EXPONENT C_SUBFUNCTION
%token CHAR INT

%left COMMA
%right ASSIGN INS_ADD INS_MUL INS_XOR INS_XNOR
%left LOR
%left LAND
%left EQ NEQ
%left LEQ GEQ GREATER LESS
%left OR  MINUS DELAY TRISTATE WIREOR AT
%left DIVIDE AND MOD
%left XOR XNOR
%left EXPONENT
%left NOT INC DEC BUFFER SCHIMIT RISE FALL HIGHVALUE LOWVALUE ASYNC

%%

id        : IDENTIFIER
          ;
int       : INTEGER
          ;
arr_id      : id index_description_list
            ;
index_term : LBRACKET expression RBRACKET
```

```
        ;
index_description_list : index_term
                | index_description_list index_term
                ;
expression : primary
        | NOT expression
        | INC lvalue
        | DEC lvalue
        | lvalue INC
        | lvalue DEC
        | BUFFER  expression
        | SCHIMIT expression
        | RISE    expression
        | FALL    expression
        | HIGHVALUE expression
        | LOWVALUE expression
        | TRISTATE expression
        | expression ASYNC expression
        | expression XOR expression
        | expression XNOR  expression
        | expression AND   expression
        | expression DIVIDE expression
        | expression MOD   expression
        | expression OR    expression
        | expression MINUS expression
        | expression EXPONENT expression
        | expression DELAY expression
        | expression WIREOR expression
        | expression AT    expression
        | expression LESS  expression
        | expression GREATER expression
        | expression GEQ   expression
        | expression EQ    expression
        | expression LEQ   expression
        | expression NEQ   expression
        | expression LAND  expression
        | expression LOR   expression
        | lvalue ASSIGN expression
        | lvalue INS_XNOR expression
        | lvalue INS_XOR expression
        | lvalue INS_ADD expression
```

```
        |lvalue INS_MUL expression
        |expression COMMA expression
        ;
primary   : id
        |int
        |LPAREN expression RPAREN
        |NO_SIGN id LPAREN expression RPAREN
        |id LPAREN RPAREN
        |arr_id
        ;
lvalue    : id
        |arr_id
        ;
compound_statement : LEFT_CURY_BRACKET statement_list RIGHT_CURY_BRACKET
                   ;
statement_list : statement
               |statement_list statement
               ;
statement : compound_statement
        |expression SEMICOL
        |IF LPAREN expression RPAREN statement
        |IF LPAREN expression RPAREN statement ELSE statement
        |FOR LPAREN expression SEMICOL expression SEMICOL
          expression RPAREN statement
        |BREAK SEMICOL
        |CONTINUE SEMICOL
        |CLINE statement
        ;
declaration_list : declaration_statement
                 |declaration_list declaration_statement
                 ;
declaration_statement : name_statement
                      |parameter_statement
                      |inorder_statement
                      |outorder_statement
                      |variable_statement
                      |subfunction_statement
                      |subcomponent_statement
                      |iif_variable_statement
                      |c_subfunction_statement
                      ;
```

```
name_statement : NAME COLON new_id SEMICOL
               ;
parameter_statement : PARAMETER COLON id_list SEMICOL
                    ;
inorder_statement : INORDER COLON id_list SEMICOL
                  ;
outorder_statement : OUTORDER COLON id_list SEMICOL
                   ;
variable_statement : VARIABLE COLON id_list SEMICOL
                   ;

iif_variable_statement : PIIF_VARIABLE_TOKEN COLON id_list SEMICOL
                   ;
subcomponent_statement : SUBCOMPONENT COLON id_list  SEMICOL
                       ;
subfunction_statement  : SUBFUNCTION COLON id_list SEMICOL
c_subfunction_statement  : C_SUBFUNCTION COLON function_list SEMICOL
function_list : function_term
              | function_list COMMA function_term
              ;
function_term : CHAR AND new_id LPAREN RPAREN
              | INT new_id LPAREN RPAREN
              ;
id_list : new_id_term
        | id_list COMMA new_id_term
        ;
new_id_term : new_id
            | new_id  index_description_list
            ;
new_id : NEW_IDENTIFIER
       ;
prog : de_list compound_statement
     ;
de_list : declaration_list
        ;
%%
```

APPENDIX B
Component Query Language

(This appendix was written by Gwo-Dong Chen)

1. Component Query Language (CQL)

Component Query Language (CQL) is the language for the Intelligent Component Database system (ICDB). CQL consists of three kinds of commands: (1) component query, (2) component generation, and (3) component list management.

ICDB can generate micro architecture level components such as registers, counters, ALUs, etc. for given set of attributes and constraints. Each component may have several different implementations.

A user can ask ICDB to generate a component by supplying a description of the component and constraints such as delay time and area. First, a component specification should be created. Then, He or she can use the component generation command to generate components.

The component query commands can be used to query the database about the components that can execute a required set of functions. Delay and area information of generated components can also acquired by these commands. A component connection query is provided to let a user get port name and control code of generated components.

A lot of components are generated during the design process. ICDB provides the component list commands for users to maintain these components easily and to prevent the regeneration of a component.

2. Terminology

A **function** is

a logic operation (AND, OR, NOT, NAND, NOR, XOR, XNOR),
an arithmetic operation (ADD(+), SUB(-), MUL(*),
DIV(/), INC (++), DEC (--)),
a relation operation (EQ, NEQ, GT, GE, LT, LE),
a select operation (MUX_SCL select by control line,
MUX_SCG select by guard value),
a shift operation (SHL1, SHR1, ROTL1, ROTR1, ASHL1, ASHR1,
SHL, SHR, ROTL, ROTR, ASHL, ASHR),
coding functions (ENCODE, DECODE),
interface functions (BUF, CLK_DR, SCHM_TGR, TRI_STATE),
wire function (PORT, BUS, WIRE_OR),
switch box function (CONCAT, EXTRACT),
clock generator (CLK_GEN),
delay (DELAY),
or a memory operation
(LOAD, STORE, MEMORY, READ, WRITE, PUSH, POP).

These operations are defined in GENUS library [Dutt 88].

The design synthesis tools use function names to query

the component database.

A **component** is a standard specification of a circuit implementation. Usually, it is a standard circuit used in micro architecture level design. Sometimes, it may be necessary to define a special component. In these cases, a

component can be defined by the names of function it executes, I/O port names, control ports, mapping between I/O operand name of a function and I/O pin name of the component. Predefined components in ICDB are listed as follows:

Logic_unit,
Mux_scl, Mux_scg,
Decode, Encode,
Comparator,
Shiter, Barrel_shifter,
Adder_Subtractor,
ALU,
Multiplier,
Divider,
Register,
Counter,
Register_file,
Stack,
Memory,
Buffer,
Clock_driver,
Schmitt_trigger,
Tri_state,
Port,
Bus,
Wire_or,
Concat,
Extract,
Clock_generator,
Delay.

A **component implementation** is a description of a component in a representation such as VHDL or CIF. It may be a fixed component described by a VHDL structure or a parameterized component generated by a generator in the ICDB.

A **component specification** is the specification of a component. Before a user want to generate components, he or she should create a component specification first. ICDB will generate a components according to the component specification.

A **component instance** is a design that ICDB generates. A component is only a specification. When the users request generation of a component, the design generated by ICDB is called a component instance.

A **component list** is a list of component instances that are supported to a a particular designer.

3. Naming

There are some predefined names in ICDB. They are (1) function names, (2) component names, (3) I/O port names of functions, (4) I/O port name of components, (5) attribute names. The names of ICDB predefined functions and components are listed in the above section.

The I/O port names of each function are I0, I1, I2, ... for input, and O0, O1, O2 .... for output. If a function is a unary operator, then its input is I0 and its output is O0. If the function is a binary operator, then its inputs are I0 and I1 and output is O0. The input of a bitwise logic operation is I0. An I/O port may be indexed by an attribute. For example, the I0, I1, and O0 port of an ADD operation can be indexed by size. A function can define an alias name for some

input. For example, we can define Cin for I2 in ADD function. All the predefined alias name of functions are defined in GENUS [Dutt 88]. The name of clock are clk0, clk1, ...etc. If only one clock is used, then the name of the clock line is clk.

The predefined I/O port names of a component are the same as the predefined I/O port names of a function. Control lines of a component are designated by C0, C1, C2, etc. I/O port names of a component may have an alias name also. For example, a comparator component has O0, O1, O2, O3, O4, O5 output ports. They have the alias name as OEQ, ONEQ, OGT, OLT, OGEQ, OLEQ. The alias name of the predefined components in ICDB are list in pp.12-pp.14 of [Dutt 88].

There are some predefined attribute names of a component. They are (1) size (input bit length), (2) input_latch, (3) output_latch, (4) input_type, (5) output_type, and (6) output_tri_state. The input_latch/ output_latch attribute describes whether a component has a input_latch/ output_latch. The input_type / output_type attributes describes the input/ output of a component is high active or low active. The output_tr_state describes whether a component has a tri_state buffer on the output port.

The name of a component implementation is assigned by the designer. The I/O port name of a component implementation may be (1) a parameter, (2) a

alias name or (3) default name as it is in a component.

The name of a component instance can be defined by the user or can be generated by ICDB. Users can query ICDB about this component instance by that name. A user can also assign the I/O and control name of a component instance when he requests ICDB to generate a component instance.

4. CQL syntax

To use ICDB in a C program, a user should use the function call ICDB(). The parameters of an ICDB() call are passed in the format as shown below.

ICDB("command description string",variable names);

If a users want to use ICDB interactively, only the command description string is required. ICDB provides a interactive user interface program. A user can enter the command description string and the user interface program will call ICDB and display the result on the screen.

The command string has the following format:

left hand side(a name):right hand side(a value)

Terms are delimited by a semicolon (;). The left hand side name identifies the attribute name (keyword), and the right hand side specifies the attribute value.

If the right hand side value is supplied by a C variable, it should have a variable description. The variable description has two parts. The first part is to indicate that the variable is to be input to ICDB (denoted by %) or to be

output from ICDB (denoted by ?). The second part is to indicate the data type of the variable. The following variable types are allowed.

symbol	type
d	integer
d[]	array of integer
s	string
s[]	array of string
r	float
r[]	array of float
f	file name

Communication between a C program and ICDB is through the variables in the second part of a variaable description. The variable/value pairs are matched by the order in which they appear in the command string.

An example of an interactive query is shown below.

```
command:request_component;
component_name: Adder_Subtractor;
size: 4;
strategy: fastest;
component_instance: ?s
```

The **command:request_component** indicates that the command is request_component. It requests the fastest adder (denoted by **strategy: fastest** and **component_name: Adder_Subtractor**). The size of this adder is 4 (denoted by **size:4**). The result of this query is returned as a string array (denoted by **component_instance: ?s[]**). The following is the same query as the above query except that it is used in a C program.

```
#include "ICDB_struct"
char *comp_name;
int bit_length;
char *instance;
comp_name=(char *)mallac(strlen("Adder_Subtractor")+1);
strcpy(comp_name,"Adder_Subtractor");
bit_lenth=4;
.....
ICDB("
command:request_component;
component_name: %s;
size: %d;
strategy: fastest;
component_instance: ?s",
comp_name,bit_length,adder_instance);
```

It requests an Adder_Substrator. (denoted by **component_name: %s** and **comp_name** as the first name in the variable names list). The size of this adder is specified in a C integer variable **adder_size** (denoted by **size: %d** and **adder_size** is the second name in the variable names list). ICDB will generate a component and put its name in the **adder_instance** variable (denoted by **component_instance: ?s**).

Each program that calls ICDB should include the **ICDB_struct** into his program. Sometimes ICDB returns a string array or an integer array to the calling program. A string array in ICDB is stored as follow:

```
char **str_array;
str_array=(char **) malloc((no_of_array_element+1)*sizeof(char *));
.....
/* The last element of the string array is a NULL pointer.*/
str_array[last_array_element+1]=NULL;
```

An integer array in ICDB is stored as follow:

```
int *int_array;
int_array=(int *) malloc((no_of_array_element+1)*sizeof(int));
.....
/* The last element of the int array is a constant        */
/*        END_OF_INT_ARRAY                                */
/* This constant is defined in ICDB_cql_constant file.    */
/* #define END_OF_INT_ARRAY -99999                        */
int_array[last_array_element+1]=END_OF_INT_ARRAY;
```

The interactive query and the ICDB() function call are the same except that the input values of a interactive query are constants while those of the ICDB() function call in a C program are C variables. For easy reading, we show all the examples in the interactive format.

5. Component query

Component query commands of ICDB include (1) function query, (2) component query, (3) component instance query, and (4) connect a component.

In behavior synthesis, a tool may start to use a function query to get the possible component implementations in ICDB which can execute a function. Then, he can use the component query to get the functions executed by each component implementation. Thereafter, it can build a function to component implementation mapping network. According to this, a synthesis tool can do the function allocation and merging.

Then, the synthesis tool can ask ICDB to generate a component instance by prividing the function name, attribute values, and constraints. ICDB will try to generate a component instance which meets these constraints. However, the constraints may be too tight. ICDB will relax the constraints and generate the component instances.

Users can use the component instance query to get timing and area estimates to do state binding, operator binding and merging.

5.1. Function query

A function is a basic operation in a micro architecture level. It is a node in the data flow graph. A synthesis tool can use this query to get the components or component implementations that execute this function. And, it can use this query to find possible way to merging functions in the data and control flow graph by supplying multiple functions to get components that can execute multiple functions.

keyword	value type
function	list of string
component	list of string
implementation	list of string

For example, we want to find component implementations that can execute both ADD and SUB functions. We can use the following query.

```
command: function_query;
function:(ADD,SUB);
component:?s[]
```

We can also directly query for component implementations that can execute ADD and SUB functions.

```
command: function_query;
function:(ADD,SUB);
implemntation:?s[]
```

5.2. Component and implementation query

In the allocation phase, a synthesis tool may want to get the functions that a component or a component implementation can execute. This information is required for merging two or more function nodes into one component.

keyword	value type
component	string
implementation	string
function	list of string

An example of component query is shown below. A user wants to know how many functions the **alu-4** implementation can execute.

```
command: component_query;
implementation: alu-4;
function:?s[]
```

5.3. Component instance query

After a user ask ICDB to generate a component instance (see command in next section), he can use this command to query the area and delay estimation of these component instances. A user can also use this command to query the functions that a component instance can execute.

Keywords of this command are listed below.

keyword	value type
instance	string
function	list of string
area	string
delay	string

Currently, ICDB return a list of all possible area estimations for all possible aspect ratios of a component. Each record on the list is of the following format:

strip = *** width = *** height = *** area = ***

where *** represents a number. For example, a component has 4 subcomponents. This component can be layouted in 1 to 4 strips. ICDB will return area estimations as follows.

strip = 1 width = 12 height = 7 area = 84
strip = 2 width = 8 height = 9 area = 72
strip = 3 width = 8 height = 11 area = 88
strip = 4 width = 5 height = 13 area = 65

ICDB will return the area of this component of each configuration. And, the

delay information is also stored in a string file which includes delay time for each output port, set up time for each input port, and minimum clock width. The format of this file is as follows:

```
CW   ****
SD   %%% ****
WD   $$$ ****
```

where *** is a real number ,$$$ is a name of output port, %%% is a name of input port, CW means clock width, WD is the delay, SD is the setup time. In the following example, a component instance **adder4** is generated by ICDB. If a user want to know the functions it can executes, the following query is inputed:

```
command:instance_query;
instance:adder4;
function:?s[];
```

The following example showns a query for getting the functions of a **three_bit_up_down_counter** and its delay.

```
command:instance_query;
instance:three_bit_up_down_counter;
function:?s[];
delay:?s;
```

A possible ouput of this query is shown below.

```
functions:
                    LOAD STORE INC DEC
delay:
                    CW 20.346666
```

WD O[2] 5.580000
WD O[1] 12.330000
WD O[0] 7.840000
SD UPDOWN 100

The **three_bit_up_down_counter** component has three output: O[0], O[1], and O[2]. The estimated worst delay from any input to output O[2] port is 5.580000, to O[1] port is 12.330000, to O[0] port is 7.840000. The minimum clock width allowed for this component is 20.346666. The input port **UPDOWN** has a set up time 100.

5.4. Component connection

After the allocation phase, a synthesis tool may bind components to operations. For that purpose it can use the connection query to obtain the information of how to connect this component instance.

keyword	value type
instance	string
connect	string

ICDB will return a string which describes how to connect this component instance. The format of this string is as follows.

```
## function function_name_0
I/O_port_name_of_function is I/O_port_name_of_instance
....
** control_port_name_of_instance value
...
## function function_name_1
```

....

The following example shows how to get the connection information from ICDB for the **add_sub_4** instance.

```
command:connect_component;
instance:add_sub_4;
connect:?s;
```

The result will be stored in a string. In case of interactive query, the output will be printed as follows:

```
## function ADD
I0 is I0
I1 is I1
Cin is Add_Sub
O0 is O0
**  Add_Sub 0
## function  SUB
I0 is I0
I1 is I1
Cin is Add_Sub
**  Add_Sub  1
```

6. Components generation

To get the area and delay information of an implementation, a generate component command should be issued to generate a component instance. The information of this component instance can be acquired by using the ICDB component query.

6.1. Component description

There are three types of component generation in ICDB: (1) from a given component name or a component implementation name and its attribute values, (2) from an IIF description, and (3) from a VHDL net list, in which components may be a complex logic gate or a component instance of ICDB. For all the three types of component generation, a user can also specify (1) delay time constraint, (2) area constraint, (3) the generation target level, and (4) the name of the design and I/O port name.

Attributes (e.g. word length, input_latch presence, high or low input activation, etc.) of the required component instance should be given; otherwise the default values are assumed.

A component implementation can be input by IIF to be used in a particular design. For example, a control unit can be expressed in IIF, and use this command by supplying a IIF description to generate a component instance from it. Then, the time and area estimation of this control unit can be obtained. The layout of this control unit then can be generated. After a behavioral description has been mapped to a micro-architecture structure, the floor planner can use this command by supplying a VHDL net_list to generate a component instance for different partition and clustering of the design. Then, it can use the component query to get the area estimation of them. The component used in

this VHDL net list can be a basic core (complex gate or logic gate) and component instances in a component list.

Several steps are involved in generating the layout from the component implementation description. For example, in the component generation path from IIF description, two steps are involved: MILO accepts IIF and generates a net_list built from logic level components, while LES takes the net list and generates a layout. Because the layout generation of LES may take a long time and the area information of the layout can be produced by the area estimator, a user may assign the **target** level at which the component instances are to be described.

In each step of generating a component, there may be some decisions or constraints specified. For instance, a delay time constraint (from any input port to output port under output port load conditions) and output load information is required for MILO to do logic optimization and technology mapping. There are three types of delay constraints, combination delay (**comb_delay**), set up time (**seq_delay**), and clock width (**clk_width**) in the sequential circuit. A combination delay is described by a triple (output port, delay constraint, output port load). A set up time is described by a tuple (output port, set up time). The combination delay and set up time can be specified by an integer which is the worst delay time. From the net_list of technology specific logic gates in

terms of layout, the pin position and no of strips of the layout should be given to LES to generate the layout.

A user can use **strategy** to specify the timing constraints. When strategy is **fastest**, a zero delay time is supplied to MILO. MILO will generate this component as fast as possible. When strategy is **cheapest**, a 1000 delay time is assumed which results in a minimum area circuit.

A user can also specify the I/O port names of this component. The specification is a list of terms. Each term has a default I/O name of the component as left hand side, and the required I/O port name as right hand side.

Keywords of this command are listed below.

keyword	value type
command	request_component
component_name	string
implementation	string
instance	string
IIF	string
VHDL_net_list	string
attribute	list of term
target	string
comb_delay	list of delay term / integer
seq_delay	list of term
clk_width	integer
pin_position	string

naming	list of term
strategy	string
instance	string / list of string

6.2. Generation from a component specification

In this command, a user can give a **component** name that ICDB will used to search for and generate component instances from all component implementations in ICDB. He can also assign a particular **component implementation.**

In the following example, the user already chose the **ripple_carry_adder** implementation. He wants an eight bit (denoted by **attribute:(size:8)**) adder with delay to O[7] and Cout less than 20 ns (denoted by **comb_delay:(O[7]:20,Cout:20)**).

```
command:request_component;
implemntation:ripple_carry_adder;
target: logic;
function:(add);
attribute:(size:8);
comb_delay:(O[7]:20,Cout:20);
instance:?s
```

A user can also specify the component name, the functions required, and the attribute value to request ICDB to generate instances.

```
command:request_component;
component_name:counter;
target: layout;
```

```
attribute:(size:4);
function:(load,up,down);
comb_delay:10;
seq_delay:10;
instance:?s
```

The above query specifies the worst delay from any input to output is 10ns (**comb_delay:10**), and set up time (**seq_delay:10**) of any input port should be less than 10 ns. The instance will call LES to generate the layout. A user can specify the strategy instead of specify the delay constraints. IN the following example, ICDB is asked to generate the fastest instance.

```
command:request_component;
component_name:counter;
function:(load,up,down);
strategy:fastest;
instance:?s
```

6.3. Generation from VHDL net list

The floor planner may want to partition a design to make the best floor plan. It may require the delay and area information of a clustering of some component instances in the design. A VHDL net_list can be generated for that clustering object and can be input ICDB to get information about area and delay. Thus, the floor planner can try different partitioning alternatives and get an estimate of the result.

An example is shown in the following. The subcomponents of cluster-1 are ICDB component instances of a design.

```
command:request_component;
VHDL_net_list:cluster-1;
strategy:fastest;
instance:?s
```

7. Component list management

The component list commands includes (1) strat a design, (2) start a design transaction, (3) put into component list, (4) end a design transaction, (5) end a design. Users use **start a design** to create a component list. Whenever users use ICDB to do design, they should begin with the **start a design transaction** command. In the design transaction, they can use **put into component list** to keep the required component instances. When a design transaction is ended by the **end a design transaction** command, the components instances are all deleted except those in the component list. After the design is completed, users use the **end a design** to delete the component instance in the component list.

The followings are the keywords of these command.

keyword	value type
command	"start_a_design"
design	string

keyword	value type
command	"start_a_transaction"
design	string

keyword	value type
command	"put_in_component_list"
design	string
instance	string

keyword	value type
command	"end_a_transaction"
design	string

keyword	value type
command	"end_a_design"
design	string